Unified Meta-Component Model Specification Editor

TR-CIS-0330-04

Richard M. Neidermyer April 20, 2004

A Project Submitted to the Faculty of Purdue University
for the Degree

of

Master of Science

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
20 APR 2004		**00-00-2004 to 00-00-2004**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Unified Meta-Component Model Specification Editor	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Indiana University/Purdue University,Department of Computer and Information Sciences,Indianapolis,IN,46202**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**148**	

To my wife, Pamella

ACKNOWLEDGEMENTS

The opportunity for me to participate in the M. S. program at the Department of Computer Science and Information Science of Indian University – Purdue University Indianapolis (IUPUI) has been an excellent experience. It has provided a formative educational experience that I can use as a basis of knowledge and guidance throughout my life and career goals. I would like to offer my sincere gratitude to everyone who has guided me through this process and has made it a positive experience.

I would first like to thank those who have guided me through the project process offering their time and expertise in completing this M.S. project. I would like to express great gratitude to Dr. Andrew Olson, who has provided continuous feedback and constructive criticism at every level of research and development. Thanks to Dr. Rajeev Raje for providing the opportunity to work on the collaborative UniFrame project and his guidance throughout my degree course work. Additionally, I would like to thank Dr. Mihran Tuceryan for being a member of my project committee and his project review.

I would like to offer my gratitude to all those involved with the UniFrame project including the U.S. Department of Defense and the U.S. Office of Navel Research for their support of the project under award number N00014-01-1-0746. I also offer thanks to my student colleagues who offered feedback during the UMMSE prototyping and development stages.

Finally, I would like to thank my family for all their encouragement and support.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Neidermyer, Richard M., M.S., Purdue University, May 2004. "Unified Meta-component Model Specification Editor". Major Professors: Andrew Olson and Rajeev Raje.

The UniFrame development process encompasses many aspects of a large-scale development effort for creating distributed computing systems. The main aspect of this process is component communication translation provided by the unified meta-component model specification. It interfaces with repositories, headhunters, component developers, and system users. This M. S. Project Report addresses qualitative communication problems introduced by the vast interface model, focusing on the interface between a component developer and the multiple UniFrame environment systems. It does so by providing a creative user interface with four major objectives: a) efficient use of personnel time during specification development and maintenance periods, b) suppression and avoidance of user error c) preemptive prevention of the miscommunication of component definitions, and d) specification modification and editorial control. Need for such an application is derived by evaluation of the current state of the UniFrame system. The Unified Meta-component Model Specification Editor (UMMSE) was developed using an adaptation of the classical waterfall software development model. The product, UMMSE, provides a unique solution to address the problem of using a XML definition file to communicate data between human and computer interactions. Component specifications (definitions) are syntactically defined using the extensible markup language (XML). This language is not user friendly in a pure editor environment and requires intervention to obtain the objective of 'safe' editing. UMMSE provides this intervention by hiding the implementation of the XML syntax behind multiple layers of common user interface editing controls. UMMSE provides a user interface required by the UniFrame system to facilitate UMM specification editing and modification supplementing human-computer interaction.

1. INTRODUCTION

The process of developing new software and software-based systems has cycled throughout the history of machine computing. New ideas are realized into implementation using previous innovations as catalysts. It is often the case that the catalyst of a software technology becomes the basis of the new technology. Furthermore, multiple previously developed ideas may be migrated in part to form the new technology.

UniFrame approaches the software development process using semiformal methods and automation to facilitate the integration of distributed software components [RAJ00, 2]. Employing an interoperable framework to form a distributed computing system (DCS) using heterogeneous, distributed components is a daunting undertaking. To achieve this, many base technologies will be merged with new development and ideals. Since the UniFrame system is distributed across multiple platforms and because it must provide varying interfaces of interoperability, it is necessary that communication channels exist.

A specific communication language that is employed uses the extended markup language (XML). Unified Meta-component Model (UMM) specifications use this language to describe software component interfaces [RAJ01, 2.2] that provide the necessary means of communication between various entities of the DCS. Additionally, a specification shoulders the additional burden of conveying human-defined, natural language concepts to a network of computing machines. This is incurred during the deployment stage of a developed software component to the UniFrame environment.

Accomplishing the goal of interoperability between two or more independently developed software components consists of multiple high-level processes. Initially, a software designer must identify the software components to be used. This process is serviced by the UniFrame Resource Discovery System (URDS), which focuses on the automatic discovery of components available for integration into the domain of

interdependent subsystems [SIR00, 2]. URDS uses headhunter components to actively read UMM specifications, registering them in appropriate repositories. Therefore, headhunter services must have a method of UMM categorization and retrieval so that those searching for particular domain components can effectively gather prospective components for integration.

Communication between domain builders, components, the URDS, and headhunter services depends on correctly identified and syntactically correct UMM specifications. It is believed that for this process to operate effectively, appropriate specification maintenance tools must be provided to software component developers. Such tools optimize developer time by providing methods for efficient specification modification. Additionally, failure resistant system communications are realized by the use of common generation tools.

1.1 Problem Definition and Motivation

Interaction between domain builders, components, URDS, and headhunter services is dependent on correctly identified and specified UMM descriptions. The UMM specification must correctly represent its component maintaining syntax and content agreement using a UniFrame defined standard format. Due to these complexities, it is imperative that a UMM standardized specification development kit be available to component developers and organizations that define component-based systems. UniFrame Meta-component Model Specification Editor (UMMSE) has been developed to provide a unique user interface and supporting backend integration to satisfy these needs.

Presently, a software component developer creates a specification by using standard text and XML editors. This is a three-step process consisting of: a) creating a new XML document, b) editing the document with content specific to the component and agreeable in UniFrame knowledge and terms, c) and syntactically correct considering both XML syntax and the specific data format required for UniFrame interoperability. This manual process has many faults that aid the possibility of creating an incorrect UMM specification, decreasing the efficiency of the development process. Furthermore,

a UniFrame system populated with a large proportion of incorrect specifications would limit the success of the technology and foster failure. This M. S. Project Report identifies three problems related to the human computer interaction (HCI) of the UMM specification model and the motivation to resolve these issues.

The first problem is the lack of a standard UMM specification editing environment for component developers. Though the editors that are currently available will perform syntax error checking, they do not provide content error checking for obvious reasons. Moreover, standard editors fail to guide the user during the input process, being purely free-form environments. In such an environment, users creating a UMM specification would be required to cross-reference with reference material to perform checks on content, agreement, and data error. An explicit specification editing environment controls specific data content within context and provides user guidance.

A general-purpose editor imposes another constraint on its user base. Inherently, because of the generality it provides, it does not offer content sensitive help or user direction. In an expanding UniFrame environment, it would be expected that not all users would be prolific in knowledge of the UniFrame system. For example, organizations may begin to hire less technically trained employees to handle the creation and maintenance of UMM specifications, thus freeing up additional software development resources to concentrate on the actual component development. Such a trend expands expected user knowledge and capabilities from all expert to variant combinations of beginner, intermediate, and expert users. To reach success in an expanding environment, the system must employ members at a greater productivity rate. This can only be achieved by the appropriate provisioning of user-friendly tools.

Using a non-proprietary, hierarchically structured language, the UMM specification offers many benefits to the UniFrame DCS development process. However, there is an accompanying drawback when introducing XML into an environment that has more than just machine entities. XML was developed to serve as an efficient and simple inter-process machine communication interface. For machines, the impact of syntax constraints as well as non-natural language does not pose any issues. Conversely, when

such mechanisms are required to work in a partnership with a human, interaction issues will arise.

It is intended that the UMMSE resolve the above-mentioned problems by implementing a UMM specification development kit. By introducing HCI ideas into the UMM specification editing process, a generalization is applied to the user base, the convenience of creation and maintenance is heightened, and an error control is gained. The combination of these factors increases the opportunity for constructing a correct UMM Specification by controlling the factors of human interaction that the UniFrame system does depend upon.

1.2 Objectives: Statement of Goals

The objectives of this M. S. Project are:

- To propose concepts of the development to hide the implementation of UMM specification from human interaction. Specifically, this conceptualization includes ideas of a GUI editor, GUI wizard, UMM specification conversion utilities, a headhunter test suite and an emulator.

- To research and analyze the current format and data content of the UMM specification. Based on this analysis, formalize a version 1.0.0 Document Type Definition (DTD) that represents the structure, content, and restrictions of a UMM specification XML document.

- To develop a software application with GUI support that implements the GUI editor and the GUI wizard. This application is to be a fully functional UMM specification editor based on the Windows operating system (OS) (although OS dependencies are to be as minimal as possible).

- Provide feature analysis of UMM specification tools and utilities that are conceptualized by this M.S Project Report though not implemented. These ideas are presented as possible topics for future work when their reasons for exclusion from this report may be overcome.

The approach used in this M. S. Project to realize the above stated goals is as follows:

- Using a limited cycle waterfall software model, complete the following steps in succession: requirement specification (user and functional), design, implementation, and integration. Following the integration of UMMSE ascertain its functional quality and compose a change requirement listing to be used in the next maintenance cycle (not acted upon in this project).

- The creation of a programmer's guide based on the final implementing code base. This guide is to be included with the final code base deliverables: application source code, compilation environments, and compiled executable.

1.3 Statement of Related Work

Works related to this project are detailed throughout this report. They vary in content and fall into three categories. First, UniFrame related research that isolates certain aspects of the UniFrame project and related sub-projects, specifically research that identifies UMM specification design and analysis. Second, HCI related research, from various eras of machine computing that relate to the use of GUIs and how computers interact, specifically analyzing the field's past successes and failures of implemented conceptualizations. Lastly, the third category of research consists of implemented applications, their implementing code, help documentation, and editorial reviews.

Sections 4.3.1 and 4.3.2 detail the process of developing the UMM specification. The development is based on the previous specification work of the *Formal Specification of Components in UMM* [RAJ01] and *The UniFrame System-Level Generative*

Programming Framework [HUA00]. Each of these works offers insight into the content of a UMM specification, forming the basis of its formalization.

HCI topic research used in this report focuses on data presentation and the interaction of users with that data. Graphical user interfaces of the UMMSE are based on both forms and dialogs. Section 5.3.1 incorporates research by Booth [BOO00] in determining which interface types are appropriate for UMMSE interfaces. In the development of mechanisms for providing contextual information previous work by Berkum [BER00] that analyzes the use of navigation explorer bars is referenced.

The design phase of UMMSE's GUIs introduces multiple related works. This research is based on previously developed applications to perform functions similar to those in UMMSE. Section 5.3.1.5 reflects upon Pierlou Visual XML [PIE00] and IBM's Alphaworks Xeena [IBM00] to justify the choice of visual controls used in development. Additionally, section 5.3.2.3 references the interaction controls of XP™ Network Setup Wizard and Microsoft Visual C++™ in developing Unified Meta-component Model Specification Wizard (UMMSW).

1.4 M. S. Project Report Contributions

The contributions of this M. S. Project Report are:

- Provides a fully functional Windows application that facilitates editing capabilities for UMM specifications. The application implements document style commands and form field editing encapsulated in a standard editing style application. Source code is provided to facilitate future improvement, feature support, and operating systems porting.

- Provides a documented formalization for an initial version of the UMM specification. Documentation includes previously researched data fields and constraints as well as data fields introduced by this M. S. Project Report. Also, a deliverable DTD file is provided upon which future research can build.

- Proposes concepts of future UMM specification tools and utilities, incorporating the topics of automation and version control.

- Bridges the gap between component development and system awareness of components by elimination of unknowns and incongruence of multiple UMM specifications. Future implementations of UniFrame entities that depend on the UMM specification definition will be able to user it has a centralized, and properly structured to use as a reference. Additionally, the convenience of quickly generating formal specifications for test and implementation purposes provides efficiency to the process.

1.5 Organization of This M. S. Project Report

This M. S. Project Report is organized into seven chapters. This chapter provided an introduction to the problem statement, objectives of this report, basis and contributions of this report, as well as the UMMSE application. Chapter 2 explains the software development model used, inclusive of its advantages, disadvantages, and reasoning of its selection. The next four chapters follow the UMMSE through the initial development cycle. Chapter 3 addresses its user requirements and usability expectations, idealizes three user types, and executes a survey of prospective UMMSE users. Chapter 4 addresses its functional requirements, introduces the main software modules, and formalizes the UMM Specification upon which proper UMMSE operation is dependant. Chapter 5 develops the application design, focusing on requirement integration, advantage / disadvantage analysis, and reasoning of implementation choice for GUI and data components. Chapter 6 discusses the design of implementing code, concentrating on data objects, graphical objects, and operating system abstraction. Finally, the M. S. Project Report concludes with an evaluation of the implementation and a proposal of future work in Chapter 7.

2. DESIGN METHODOLOGY

This chapter presents the software design model used to develop the Unified Meta-component Model Specification Editor. Section 2.1 identifies development limitations assumed by the project. Section 2.2 discusses the general form of the waterfall life-cycle model, including: requirements, specification, design, implementation, integration, and change verification phases. Deviations from the traditional waterfall model for adaptation to the UMMSE project are identified in section 2.3.

2.1 Project Limitations

Any application design and implementation process encounters particular guidelines and limitations that affect the project. Because this project is implemented within the realm of a M.S. Project, certain limitations are imposed on the development process. The purpose of this section is to identify these limitations and define their impact on the implementation of the development process.

The primary limitation of this project is the personnel count, which in the case of this application development effort is one developer. Due to this, one person must implement all phases of the software development model, resulting in a linear requirement. Since parallelism effort is minimized in this environment the development cycle is drawn to its greatest length. In combination with a deadline (project completion date), this creates a strain on the model. Therefore, as it will be noted in section 2.3, certain aspects of the complete software development model will need to be shortened, or eliminated entirely, in order to meet the project requirement deliverables.

Process adaptations to account for the limitations are designed to achieve an appropriate medium between personnel constraints and the effectiveness of the resulting product. The waterfall life-cycle software development model provides the proper basis

to accomplish this task. Its granularity and linear qualities are appropriate for the adaptation requirement of this project.

2.2 UMMSE Life-Cycle Model

The waterfall life-cycle model is a classical software development approach. Its name is derived from the visualization of the approach where each phase falls in to the next phase, at which point the progression may fall further or return (churn, splash) back to the previous level. This project bases its development model on the waterfall life-cycle model as specified in Classical and Object-Oriented Software Engineering [SCH00]. There are five major phases of the waterfall life-cycle model: requirements, specification, design, implementation, and integration. Each phase incurs the tasks of initial documentation, execution, review, and iteration.

Documentation is a key characteristic of this model [SCH00, 69]. Initially, it brings guidance, intent, and clarification to tasks at hand; iteratively, it brings change control and analysis to the process, and at completion, it provides user documentation and finalization to the project. After completing initial documentation for a phase, the phase task must be executed. This may include requirement and specification analysis, or it may require actual implementation of functional specifications. The third step of each phase provides a review of the executed task. During the design phases, this process consists of verifying the produced design documents. For development phases this process consists of software analysis and testing. Both processes use software quality assurance (SQA) groups to perform the review [SCH00, 65]. It is the job of the SQA group to identify faults in design and implementation as well as to make recommendations for further improvement. The fourth phase, iteration, uses these results as its catalyst.

Iteration is the foremost important characteristic of the waterfall life-cycle model. Each phase of the model (less the first) has two exits paths. The first path leads to the next step and is representative of completion of the current phase task. The second path recants to the previous phase providing opportunity for design and implementation

improvement. This quality is a key reason for selection of this model for this project because of its provision for inevitable error.

Finally, the model includes three additional phases that complete its full life cycle. Following final integration and testing, the model enters an operations mode. In application, this phase is often referred to as an initial or controlled release of the software. During this phase project management can proceed on multiple paths. It may determine that faults still remain within the software in which case the next phase is to return to the appropriate phase to resolve the faults. Feedback from those using the software may indicate desire for feature enhancements. In this case the model will return to the specification phase specifying the requested enhancements that are to be merged into the implementation of the application. The last possible fulfillment path in the model is the retirement phase. As the nomenclature suggests, this phase represents the maintenance end of life for the software application at which point development maintenance will cease and the product is retired.

The development of UMMSE is based on the waterfall life-cycle model for reasons discussed in Section 2.4. The five major phases were adopted in their native linear order to formulate the direction of the project. Though a model specification can provide the essential structure of any development project, it is important to note that each project must also derive its own relationships within the model. The following describes the explicit use of each phase within the UMMSE life cycle.

2.2.1 Requirements

This phase formulates the system and user (client) requirements of the software application. UMMSE implementation of this phase is accomplished by first performing a user analysis to identify the expected user base for the application. Following the establishment of user requirements, analysis is performed to determine the usability of the requirements. Each of these steps is detailed in chapter 3.

Usability specifies user goals that are then formulated into detailed functional specifications during the specification phase. User and usability requirements are

influenced by human-computer interaction (HCI) concepts. An additional function of the requirements phase is to pinpoint HCI considerations that are determined to be important to UMMSE users. Introducing HCI concepts during this phase ensures that they will be developed further in subsequent phases.

2.2.2 Specification

The UMMSE implementation blurs this phase with the requirements phase because of the application's tight relationship to its user base. However, this phase is uniquely identified for providing the specific feature requirements (feature specifications) of the application; derived from analysis of usability goals. At the completion of this phase, specification documents exist that underline what the application *does* (Chapter 4) [SCH00, 67]. This project also establishes the definition and format of the UMM specification as this basis must be known before application design.

2.2.3 Design

The design phase begins the process of determining how a developer (or team) is to accomplish the goals set forth by the specification. This phase serves two important purposes. First, it provides the development documentation that is used by a team of developers to coordinate agreement among development modules. Though not a factor in this project, failing to perform this function can have major ramifications on developer time and cost of a project. Second, this initial design of an application often brings forth fallacies of the feature specification, discovering incompleteness, contradictions, and ambiguities [SCH00, 67]. When this occurs, the waterfall development model specifies that the process should return to the specification phase to further pursue correctness.

UMMSE implementation utilizes the design phase in three different instances and iterations with the implementation phase. Focusing on the application GUI, the first design incorporates the visualization of the application and its three major GUI modules. The second design phase realizes the conceptualization of UMM specification within the

application space, creating the layered data approach for data abstraction. This design includes defining the abstraction interface between pure specification data and the visual representation of that data to the user. Transformation of the physical operating system specification file (XML specification) to and from logical application data representation constitutes the third instance of this phase.

2.2.4 Implementation

As its name suggests, the implementation phase provides a concrete implementation of the specified software application. During the development of UMMSE, this phase is closely related to both the design and integration phases. It is during this process that the concrete implementations of the UMMSE GUI (editor, wizard, and multiple document application), specification data representation and conversion, and XML file formatting and parsing are developed. Each implementation is combined with multiple iterations of its bounding phases to adjust designs due to implementation issues, as well as to correct implementation due to integration issues.

2.2.5 Integration

In its strict form this phase performs concrete application module integration with related software components and its integration environment. UMMSE integration does not adhere to a strict interpretation due to those limitations noted in section 2.1. Therefore, this phase merges responsibility with the implementation and operations phases of the waterfall software development model. At completion of each concrete module implementation the developer first reviews it; with acceptance it is then presented to a review team (project committee). Review recommendations are then fed into complementing iterations of the implementation phase. Within this project space the integration phase also performs the function of an operations phase.

2.2.6 Change Verification

After acceptance of the application by the client during the operations phase, the development process enters its change verification phase [SCH00, 68]. This phase instantiates the maintenance period for the life of the application that may consist of software corrections (bugs), and adding additional feature support. A change management control is an important part of this phase, which cycles feedback to all phases of the waterfall model (appropriate phase for bug correction, or specification phase for a new feature). This project does not address this phase of the development process. Ideally, after true UniFrame integration and execution of the operations phase, future research and development will comprise the change verification phase.

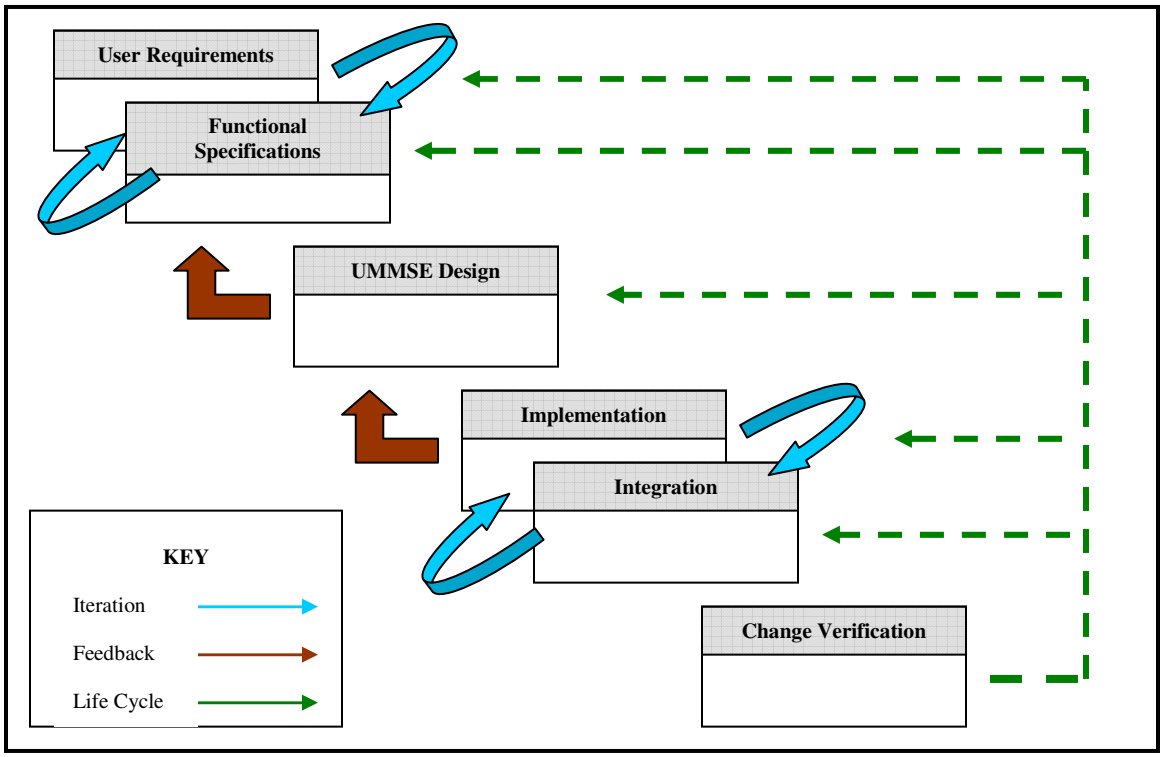


Figure 2.1. UMMSE Waterfall Model

2.3 Model Adaptation

A clear advantage provided by the selection of the waterfall development model is its adaptive characteristic. The strict definition of the process can be modified with varying degrees without changing the resulting impact of the project model. However, adaptation does lead to the possibility of minimizing other advantages of the model; though in an ideal fashion disadvantages may also be minimized. This section evaluates the impact of three UMMSE adaptations of the waterfall software development model.

This model provides the opportunity for continuous verification and testing throughout the development process [SCH00, 68]. Such fortitude brings completeness and correctness to a project, though not without the cost of valuable personnel and time – a known limitation of this project. Thus, the requirement of phase verification is limited to development review and occasional review integrations.

Documentation is another advantage of this model, and it too, brings with it the provisioning of additional resources not available to this project. Since the waterfall model in its true form is documentation intensive, it is realistic to expect that some of the documentation can be removed from the model without a significant loss of effectiveness. Specifically, documentation intended for intra-team communication is less significant in a project with a single developer. Documents to be included in the UMMSE adaptation were chosen based on their impact on two objectives: documentation 1) that supports conveying concepts and ideals of this project, and 2) that supports future design efforts and application maintenance.

As a final point, using the waterfall software development model may result in an application that does not completely meet the users' needs [SCH00, 85]. Within the design model this is intentional, and is compensated for by allowing needs to be met by incremental development cycles. The UniFrame project is currently in its conceptual phase and complete implementation and integration is not near completion. Due to this, it would be presumptive to believe that an user-end application developed at this point could completely meet user needs. Therefore, the impact of this disadvantage is minimized when adapted to the UMMSE project.

3. USER ANALYSIS

This chapter presents the user requirement phase of the Unified Meta-component Model Specification Editor development cycle. Section 3.1 identifies the three user profiles used to classify prospective UMMSE users. Section 3.2 provides UMMSE user requirement specification. Section 3.3 provides UMMSE usability requirements. Section 3.4 develops use cases identifying specific user interactions. Section 3.5 provides combination activity diagrams creating a sense of the complete interaction environment within the global space of the UMMSE application. This chapter concludes with analysis of an implemented user survey in section 3.6.

3.1 Profiles

There are three different forms of categorical knowledge of the UMMSE that can be applied to user profile analysis. All three forms are used when developing the variant user profiles. The first form concerns the user's knowledge of the UniFrame system and its components. In particular, the user's knowledge of the UMM specification is of the greatest concern for this evaluation. The second form of knowledge takes into account the user's proficiency in operating-system-based graphical controls. Specifically geared towards this project, the user profiles will be developed based on the Windows™ operating system. Finally, the third category of knowledge will be based on the user's proficiency in specific program designs and formats found in common software applications, such as a dialog wizard format.

The UMMSE project includes two major interface control formats. The main program interface consists of a common combination of main menu, toolbar, and document view. This is traditional of Windows application such as Microsoft Word™, Visual SlickEdit™, and Acrobat Reader™, among many others. The UMMSW module of the UMMSE uses the second program format. This format consists of consecutive dialog

interfaces presented to the user in a logical sequence requesting information in the process of reaching a final goal. This interface is comparable to traditional wizard controls such as Microsoft Visual C++ Class Wizard™ and the familiar install wizard used by many plug and play software packages.

3.1.1 Classifications

As in any development process, compromises must be made between different entities in order to satisfy cost, time, capability, and user constraints; user profiling is not an exception. There are three profile levels used in this process: basic, intermediate, and expert. There are specific requirements that can be associated to each level when referencing any particular form of user knowledge. These requirements are included in the analysis of module user profiles. Each classification level is evaluated with consideration for each of the three forms of user knowledge to compile an encompassing definition of profile levels in the context of UMMSE. The user classifications identified for this application are basic, intermediate, and expert.

3.1.1.1 Basic

This user ranges from a first time user to a user that infrequently accesses the application. It is expected that the user understands the basic computing principles, has a broad knowledge of the UniFrame concept, and understands that the purpose of the UMMSE is to generate a UMM specification. New employees, temporary employees, and technical managers are relative job titles that a basic user of the UMMSE may hold. It is expected that a user of this level will have to frequently query for a guidance from manuals, on-line help, or a user of greater application knowledge.

3.1.1.2 Intermediate

This user frequently accesses the application to make modifications to existing UMM specifications. In addition to the expectations of a basic user, a user of this level should be familiar with most aspects of a computing system as well as operating system dependant functionality. Support technicians, junior developers, and documentation editors are relative job titles that an intermediate user of the UMMSE may hold. A user of this level on occasion will have to access available help avenues to aid memory or to learn points of the application that had not before been encountered in typical use. However, it is expected that a user of this level is capable of seeking and understanding the help material within a reasonable amount of time without incurring project delay.

3.1.1.3 Expert

This user is considered to be a regular user of the application. It is expected that the user has worked with the application as an intermediate user for a prolonged period and possibly has formal training. The user is capable of effectively using each major feature that the application has to offer. Component developers and contracted UMM specification developers would be examples of expert users of the UMMSE application. A user of this level is capable of making decisions concerning which feature best suits any particular task, and is capable of completing that task. The user's extensive knowledge of the environment, operating system, and UniFrame system creates competency concerning application trouble-shooting (though it is optimistic that the application will run flawlessly; there are bound to be bugs that create user inconsistencies). It would be advantageous for every component development group, whether it is a team or complete company, to have an expert user at its disposal to create and supervise the development of UMM specifications.

3.1.2 Module Implications

It is not possible to design a productive user interface within the constraints of this project while satisfying each level of profile concerning each form of user knowledge. Compromises are made in an attempt to create the best solution for the proposed problem. It is the goal of this project to make the appropriate compromises while minimizing their effects on the first version of the application. Each of the two interface modules, editor and wizard, are tailored towards specific user classifications for the reasons discussed in this section.

3.1.2.1 UMMSE

- UMM Specification Knowledge – The majority of UMMSE users are expected to have intermediate knowledge of the UMM specification. A developer using the UMMSE is understood to be the developer of a specific component for the UniFrame system. The user interface direction and integrated context help is expected to be enough to allow a user of basic knowledge to learn the process, while at the same time not creating a nuisance factor for those users of expert level.

- Operating System Control Proficiency –The user interface control mechanisms can be developed in a manner where all three levels of knowledge are adequately addressed. This goal can be achieved by including a variety of controls tailored towards each user level; simple controls such as edit and check boxes for basic users, menus and toolbars for intermediate users, and keyboard shortcuts for expert users.

- Format Proficiency (Document View Architecture) – The document view architecture can be tailored to those users of intermediate knowledge of the interface format. For a basic user the interface format may not be immediately

obvious, however it is expected that through moderate exploration and use of the interface, the learning curve can be quickly overcome.

3.1.2.2 UMMSW

- UMM Specification Knowledge – By basic concept, the wizard format is naturally tailored towards a user of basic knowledge of the UMM specification. The UMMSW is intended to be a simplified version of the UMM document editor that applies additional screen instructions per input. It is expected that intermediate and expert users will only choose this interface when creating a document from scratch or to address a particular issue that has halted their progress. Since it is not mandatory to use this interface to create UMM specification, it should not interfere with typical operation by an intermediate or expert user.

- Operating System Control Proficiency – The wizard format utilizes the same control mechanisms as the specification editor. It is a simplified and segmented version of the editor interface that does not include toolbar or menu options. Therefore, it is also developed in a manner where all three levels of knowledge are adequately addressed.

- Format Proficiency (Wizard Architecture) – By basic concept the wizard format is naturally tailored towards a user of basic knowledge of the wizard format. Each window provides detailed information concerning the requested input, while maintaining as much simplicity as possible. A window represents an incremental step in the process of creating a UMM specification document. As each step is completed, the following steps are modified taking into account previously provided information if appropriate.

3.2 User Requirements for UMMSE

The purpose for developing a broad set of user requirements for this development project is to document the UMMSE application developer's perception of what is needed by the prospective users. Generating user profile guidelines and their importance concerning the variant UMMSE modules (3.1.1) completes a portion of this task. Functionality specifications (Chapter 4) and the usability requirements (3.3) are also subsidiaries of the task of creating a complete user requirement specification. This section will concentrate on the user requirements that are particular to the application's interface and interface control (user interaction aspects).

User, usability, and feature requirements may be viewed as a single specification that formulates two distinct goals. First, the specification is used as a basis for generating prospective user questionnaires. Direct user feedback is a key factor in creating a user friendly and functional interface. Near the end of the user development phase, questionnaires were generated to query for initial user feedback implementing the development process's first interaction with prospective users. Using a pre-documented requirement specification, effective user response mechanisms can be developed that minimize user involvement (and time) and maximize the quality of feedback recorded. The second goal of creating multiple requirement specifications is to create a form of direct reference concerning the project's objectives. During prototyping, development, and product finalization, requirement specifications are available for reference and initial proof of accurate implementation (though this is not a replacement for test design and execution). The potential positive impact of these two goals on the project drives the incentive for creating such a requirement directive.

The following user requirements generalize the high-level expectations of a UMMSE user. During the requirement phase they are used to conceptualize application ideas. Later in the development process (design and implementation phases) the requirements are referenced to verify agreement with the developing application.

3.2.1 Intuitive Task Direction

The main document view includes a standard application level main menu. When navigating the multiple menu selections, the user should have all the information required to make a determination for proper selection in the context of the user's intentions. To accomplish this, all elements of the menu selections must be descriptive as well as consistent with the user's innate and learned ideals. Elements are composed qualities such as titles, colors, icons (if applicable), pop-up movement, and activity variants (shade). To accomplish any particular task the least amount of user input should be required. Accomplishing this goal is important in order to limit the number of selections per menu selection. Equally important is grouping similar task selections within the same menu so that a user can properly navigate the menu using 'intuitive' instincts when the exact location of a selection is not known (or remembered).

3.2.2 Intuitive Tool Recognition

The main document view includes a standard application level toolbar. This toolbar provides a secondary method for selecting a limited set of the same tasks available by way of the main menu. It is important that the user be able to easily and accurately recognize the feature being represented by a particular icon. If applicable, the icon should be the same as that displayed on menu options. For user convenience, preferable toggle configuration of any toolbar should be available to hide a toolbar from the user's view. This is a common practice of expert users who can mentally recall all the keyboard shortcuts and would prefer to save the screen real estate for the document view.

3.2.3 Document Readability

It is imperative for each view that it adheres to three user requirements: organization, eye appeal (for easy reading), and layout maneuverability. Organization requires the document view to appropriately proportion and group specific items of the

UMM specification together. Eye appeal is a requirement of all users, though especially important for the frequent reference user [DAV00, 47]. This requirement includes sub-items such as font attributes, color, and contrasts. Consistently reading on computer screens is a common problem in general; therefore it is important that information presentation ease the strain on the eye as much as possible. Proper document maneuverability allows the user to view the complete document when it is not possible to view all information within a single screen display. The user should be able to bring off-screen data on screen without interrupting the flow of reading; this requires simple controls, smooth scrolling, and undetectable pixel refresh.

3.2.4 Document Writeability

A feature of the main document views is to provide editing capabilities for UMM specification documents. User requirements emphasized by this view consist of elements that request user input. The view should be easily navigated between the variant input controls, achieved using tab architecture. There should be an obvious division between prompts for data (application provided text), the control used for the input, and the actual user provided data. Eye appeal is as important in this view as it is during UMM document read mode, especially for those expert users who frequently create and edit UMM specifications. It is important that an indication is always provided to the user so that the current position of editing is obvious to the user. This aids in avoiding miscalculated keystrokes and unnecessary pointer relocation using mouse input or tab control.

3.2.5 Appealing Visual Program

The application's overall visual program (or *programme*), the comprehensive application space addressing visual organization [MUL00, 354], must meet multiple goals for success. Among these goals, focus control is one of the most important. When visually gazing any screen within the application, the user's focus should be directed to the proper point. Distracting embellishments, such as bright colors and active graphics,

do not provide appropriate user benefit and are to be avoided. Additionally, each interface should be properly proportioned using a balanced grid approach. A grid guides the spatial arrangement creating the consistency needed in an appropriate layout [HUM00, 304]. Each visual aspect and control should use an amount of screen real estate proportional to its importance and its need for effective control. Leverage is a concept of the visual design that identifies specific uses of screen real estate for multiple purposes. Used properly, it can create both a very functional application while maintaining a very skinny and slick interface. A proper balance of leverage should be used so not to limit functionality or user intuition, and yet gain inherent functional control and screen real estate. In applying these characteristics, the application's visual program should be simple, effective, and consistent between each interface.

3.2.6 Internal Convention Consistency

Internal consistency is a basic concept of visual interface design. Though it is often simple to maintain internal consistency of an application, it does require that particular attention be paid to details during the design. Without internal consistency, users may easily become frustrated when navigating the application, possibly interfering with the user's ability to achieve a goal. Internal conventions benefit a user throughout the entire application by taking advantage of a user's ability to learn. This is achieved by using the same conventions on multiple interfaces so that the user must learn and recall less. Also, when encountering a new interface, the user is given the benefit of being familiar with its view and controls even if its content is not familiar.

3.2.7 External Convention Consistency

External consistency is another basic concept of visual interface design. It involves understanding the user's environment (mainly the operating system) and creating a controlled visual interface that is consistent with similar applications. Broadening the learning curve is necessary for an intermediate user of the operating

system when adapting to the application. External consistency triggers a cascade effect of positive user interactions including the avoidance of initial frustrations and the ability to immediately learn application-centric features.

3.3 Usability Requirements for UMMSE

Overall user experience reflects largely on the success of an application. Therefore, it is important to pinpoint specific user experiences and determine a preferred response to that experience. During the design and development phase this information can be used to guide certain decisions that must be made. Usability requirements are designed to extend the user's requirements into short and long-term experience requirements that play a large role in the development and testing of the application. The following usability requirements are general concepts that are to be controlled and implemented during the development and design of every feature and interface. During development, if it is determined that a usability goal may suffer concerning a particular element, then adjustments should be investigated to determine what can be altered to meet that goal. Every possible avenue is explored to achieve all usability requirements. During the integration and operations phases, the application's users determine the success of the interface and its implementations. These requirements are developed in detail during the initial design phase of the application's GUIs.

The following usability requirements characterize the expected qualitative user experience to deem the UMMSE implementation successful. During the design and implementation phase these requirements are periodically referenced to verify that plans in progress are in agreement with expected usability.

3.3.1 Learnability

Many users, each of which learns in their own unique way, will use the UMMSE application. It is important that the application's controls and interface lend themselves to easing this learning process. Implementing this facet can directly affect application user

productivity by broadening learning curves and lowering education costs. Additionally, turn-around time from first user experience to productive usability can be vastly shortened. Multiple features and user requirements are used in conjunction to create a learnable environment. Throughout the process of development and implementation, it is important to maintain learnability and evaluate its current status within the application.

3.3.2 Appropriate Feedback

During use of the UMMSE, the user should never encounter a situation where the system appears to be unresponsive. Such a feeling may cause the user to close the application or become lost within its controls, possibly causing a loss of valuable input. When the user provides a form of input (or action) the system should respond with a logical acknowledgement of that action. In the case of editing a UMM specification document, the response may be as simple as repositioning the blinking cursor or drawing a character to the screen. If the application is performing an extended computation the response may need to be more complex, possibly in the form of a time interval status completion bar. A proper response indicates to the user the status of an action and requests any additional information that may be needed. A solid implementation of appropriate feedback emphasizes proper program control flow and should be transparent to the user. If feedback is disruptive to the user process then its implementation is likely inappropriate and should be changed.

3.3.3 Frustration

Throughout the implementation and integration phases it is important to identify possible points of user frustration generated by the application. All software applications have the possibility of introducing frustration into a user's computing experience. It may not be possible to identify and eliminate all frustration points. However, maximum effort should be taken to minimize the number of these points within the application. Key areas of interaction that can be focused upon during this process include program control

movements, system response, and expected application behaviors. Overall user experience and productivity are improved by implementing frustration control during the development process.

3.3.4 Text Balancing

Static and dynamic interface text listings are used to serve multiple purposes within an application to communicate instruction to the user. A balance must be maintained between the information content of the text and its presentation to the user. The goal of text is to convey a concept or direction of use to the user within a context of its purpose. This goal must be achieved in the context of the expected user profile and the application's current environment. It is important that the textual message be simple so that the least knowledgeable expected user classification gains understanding. This is a requirement to achieve the message's initial goal, which is to convey a message to the user. However, it is also important not to insult a savvier user, as well as not to waste screen real estate to display unnecessary text.

Many methods can be used to aid in determining when text should be displayed and at what level its content should be tailored towards. Using user configurable options is one method. An option can be provided that indicates the users competency (profile level), or to select what type of text prompts the user would prefer to see on a consistent basis. By employing some of these techniques and providing adequate user feedback forums, a workable balance of text control is achievable.

3.3.5 Exit Control

During typical application use differing scenarios may be encountered. Users may or may not be familiar with any given scenario; therefore, it is important that they have a sense of control over their current situation. This is important in situations where data may not have reached a point of being saved and is necessary to make sure that the data are not lost. The concept of exit control is created using internal and external

consistencies as well as conceptual directives. At any point throughout the application the user should easily be able to make an instinctive determination of ‘exit’ options such as forward, backwards, save, and exit. In the case of an unexpected situation, the user should have been provided and is aware of all possible exit methods to alleviate the situation. When executing the options, the user should have a predetermined notion of the consequences of the particular action and can therefore be provided the opportunity to make an educated decision.

3.3.6 Goal Achievement

The intent of the UMMSE application is to provide a utility of service to users of the UniFrame system. Obviously, any user of this application has a particular goal for every action as well as multiple greater goals that are often to produce tangible results. Since the UMMSE application is goal oriented, it is essential that user goals be accomplished in timely and productive fashion; therefore goal achievement is an important usability requirement. If a goal cannot be completed because its user’s progress is hindered by a poorly designed interface, then the user’s (and organization’s) experience using UMMSE may be poor. Goal-oriented use cases are developed in the next section (3.4) to emphasize the importance of this aspect. Use case descriptions and diagrams pinpoint specific actions that a user performs, analyzes how they may be performed, and indicates proper system responses. Creating, implementing, and verification testing against pre-defined use cases assure achieving a level of correctness is ascertained.

3.3.7 Input Error Detection \ Correction

A core focus of the UMMSE application goals is to create multiple interfaces that make the process of creating and editing UMM specification documents easier. Implementing proper error detection and correction opportunities is a large part of developing this focus. Throughout regular use of UMMSE, every user will undoubtedly

input information erroneously. There are two steps of counter-measures that the interface may implement to create an environment of error-free data finality. The first preempts the inputted data with attempts to direct the user properly, and to limit the input variability in the form of restrictive controls. Of course, free input control is sometimes required; therefore it is important to implement a method of error checking post user input. This is a much more complex process and may require in-depth spell checkers as well as some artificial intelligence (AI) logic, which could be accomplished in future work. Once an error is detected it is convenient for the user if the system offers some suggested choices towards correcting the error. If a concrete suggestion is not possible, then examples or approximate intentions can suit as a proper correction suggestion. User error protection is a key element for any user-friendly application. Beyond preventing erroneous results, it can also save the user time, as well as provide the user with a more pleasurable computing experience.

3.4 Use Cases and Program Controls

There are two forms of use cases developed in this section. Each form ideally represents actions and responses that will occur during typical interaction of a particular item. If multiple branches of user control or program response are possible then the use case will provide optional control paths.

The first form of use case concerns general program control and is focused on the main document view interface, in particular its menu and toolbar options. Rather than depicting interactions required to complete a specific task, this form emphasizes program flow. This develops the relationship between the multiple dialog views and the application's main menu. It also serves as a platform to describe how a user may best benefit from integrating with the multiple options provided by the UMMSE.

The second form of use case deals with specific functionalities and how a user may go about completing any particular task. This form closely depicts the features developed during the feature specification phase (chapter 5). A particular task will be outlined followed by a use case description of user interactions and system responses.

This form also serves as a platform to outline possible error conditions and how a user may be able to recover from those errors. Not all features have a corresponding use case; however, care has been taken to include features that develop a basis for a majority of interfaces and interactions.

3.4.1 Compare and Edit Multiple UMM Documents

1) User Opens UMM Document 1 Into View Using Main Menu Option
2) System Loads Document 1 Into View – Maximized
3) User Opens UMM Document 2 Into View Using Main Menu Option
4) System Loads Document 2 Into View – Hides View of Document 1
5) User Edits Field 1 of Document 2 – Keyboard Strokes
6) User Focuses on Field 2 of Document 2
7) User Presses Ctrl+Tab
8) System Loads Document 1 Into View and Focuses on Field 1
9) User Edits Field 1 of Document 1 – Keyboard Strokes
10) User Presses Ctrl+Tab
11) System Loads Document 2 Into View and Focuses on Field 2 (last known focus)
12) User Selects *View Toggle* Toolbar Option
13) System Displays *View Toggle* Sub-menu
    a. Includes Document 1 Listing
    b. Includes Document 2 Listing (with checkmark indicating current focus)
    c. Includes {others}
14) User Selects Document 1 Listing from Sub-menu
15) System Loads Document 1 Into View and Focuses on Field 1
16) Repeat: Editing, Ctrl+Tab, and Menu File Toggling While Multiple Files are in View

3.4.2 Start UMMSW To Create New UMM Document

1) User Selects “Utilities” From Main Menu
2) System Displays “Utilities” Sub-menu
3) User Selects “Launch Wizard” From Sub-menu
4) System Launches UMMSW Module At Its Opening Dialog
5) System Focuses on UMMSW Opening Dialog – First Edit Option “Component Name” is Set in Focus
6) User Enters a Component Name
7) System Sets *Move To Next Step* Button to Active

3.4.3 Edit Current UMM Document Using UMMSW

1) User Selects “Launch Wizard with Document 1…” From Main Menu
2) System Launches UMMSW Module
3) System Loads UMM Document 1 Into Wizard Edit View
4) System Opens UMMSW Dialog with Contextual Focus*
5) System Sets Input Focus on First Edit Option

*Contextual focus refers to the current field of edit in the current UMM document. This field is directly associated with a particular dialog included in the UMMSW. This link is used by the UMMSW to determine at which input field the dialog should initially focus.

3.4.4 Exit UMMSW Before Completion of New UMM Document

Note: Each UMMSW Dialog interface, except for the last, has a button to exit the UMMSW in an incomplete mode. An incomplete mode indicates that not all required UMM document fields contain valid input.

1) User Presses the Local ‘Exit’ Button
2) System Prompts User For Save Option
    a. Do Not Save
    b. Save As <filename>
    c. Save As <filename> And Load In View
3) User Selects ‘Do Not Save’
4) System Closes UMMSW Dialog – Returns Focus To Open UMM Document In View (if applicable)

(Alternate)

3) User Selects ‘Save As <filename> And Load In View’
4) System Closes UMMSW Dialog
5) System Loads Uncompleted UMM Document Into View

(Alternate)

3) User Selects ‘Save As <filename>’

4) System Closes UMMSW Dialog – Returns Focus To Newly Saved UMM Document In View

3.4.5 Open UMM Document Into View

1) User Selects “File” From Main Menu
2) System Displays File’s Sub-menu
    a. Includes ‘Open’
    b. Includes Most Recent Document 1
    c. {multiple most recent documents (max 5)}
    d. {others}
3) User Selects ‘Open’
4) System Prompts User With Standard File Selection Dialog
5) User Navigates and Selects UMM Specification File
6) System Loads Selected Document Into View – Maximized

(Alternate)
3) User Selects Most Recent Document 1
4) System Loads Selected Document Into View – Maximized

3.4.6 Save UMM Document

1) User Selects “File” From Main Menu
2) System Displays File’s Sub-menu
    a. Includes ‘Save’ (if document has been titled)
    b. Includes ‘Save As…’
    c. {others}
3) User Selects ‘Save’
4) System Marks ‘Save’ Menu\Toolbar Options as Inactive

(Alternate)
3) User Selects ‘Save As…’
4) System Prompts User With Standard Save File Dialog
5) User Navigates Directory
6) User Enters Filename
7) User Presses ‘Save’ Button
8) System Closes Save File Dialog
9) System Changes Document Title to Filename

3.4.7 Progress Through UMMSW Towards Document Completion

1) System Displays Input Fields with Minimal Descriptive Text
2) User Inputs Requested Data
3) System Makes 'Next' Button Active When Sufficient Data Has Been Provided
4) Repeat: Steps 1-3 Until All Dialogs Are Complete
5) System Makes 'Finalize' Button Active When Sufficient Data Has Been Provided
6) User Selects 'Finalize'
7) System Prompts User for Filename to Save As (if not previously selected)
8) System Closes UMMSW Dialog
9) System Loads Created UMM Document Into UMMSE Edit View
10) System Focuses on Field 1 of Document

3.5 Combinational Activities

The purpose of this section is to offer a pictorial representation of the use cases described in the previous section. Rather than providing an individual use case diagram for each task, which would be the linear approach, the purpose is best served by use of general activity diagrams that combine multiple related use cases. An activity diagram is used to show the linear and non-linear relationships between the tasks, user actions, and system responses. At each user or system action point, the multiple possibilities of control will be listed. However only the choice best suited for the depicted task(s) will actually be further propagated in the diagram.

The first combinational diagram (3.5.2) combines the application's main menu and UMM editing interface in the process of creating a new document, editing that document, and then saving the edited document. Not all control points of the editing interface are represented because the editor is a free form module. From the editing view (with a file loaded) virtually any operation within the UMMSE application can be accessed and executed. The second diagram (3.5.3) represents the process of loading a UMM document into the UMMSW interface, navigating the multiple interface dialogs, and completing the operation by returning to the UMMSE interface. This diagram explicitly shows the relationship between each dialog and the flow control of moving

either forward or backward through the input dialogs. This is important because each dialog is an encapsulation for one or multiple related input requirements; at any point in time it may be necessary for a user to close the UMMSW, return to a previously entered field, or access context help documents.

3.5.1 Notation

A unique notation is used to represent the information provided within the following two diagrams. As previously noted, the diagrams are designed to show relationships between multiple tasks incorporating user and system action points. Each diagram is centered around a multiple selector interface because it provides the initial control generation; in the first diagram the selector represents a dropdown menu of the main application, in the second the selector represents the array of choices available on each individual UMMSW dialog.

Flow sequences are depicted using unidirectional arrows. A solid arrow (blue) represents a definitive course of action of a particular task, whereas a dashed arrow (green) represents a possible course of action, though not completed within the actual diagram. Information enclosed within a box represents actions and display messages provided by the system. For example, each initial selector is boxed because it represents a message display provided by the system to the user. User actions (typically input) are enclosed within a circle. In the general case, because the application's interaction is directed towards only one user at any particular time, a circle encapsulates an action by the user in the anticipation of transition from one system message to the next.

At certain points a user may use the same input action in response to a system message that can generate multiple exit control paths. A menu is a prime example of this situation because a user can only make one input selection, however that selection has multiple choices each of which generates a different exit path. Trailing arrow lines that pass through a particular user action represent this case. There also may be situations where a single user action may generate multiple system messages that execute in

succession or in parallel. A single horizontal bar that terminates the inward path and serves as a starting point for the multiple system message paths represents this case.

3.5.2 Activity Diagrams

Tasks Represented:
Edit Multiple UMM Documents
Open UMM Document Into View
Save UMM Document

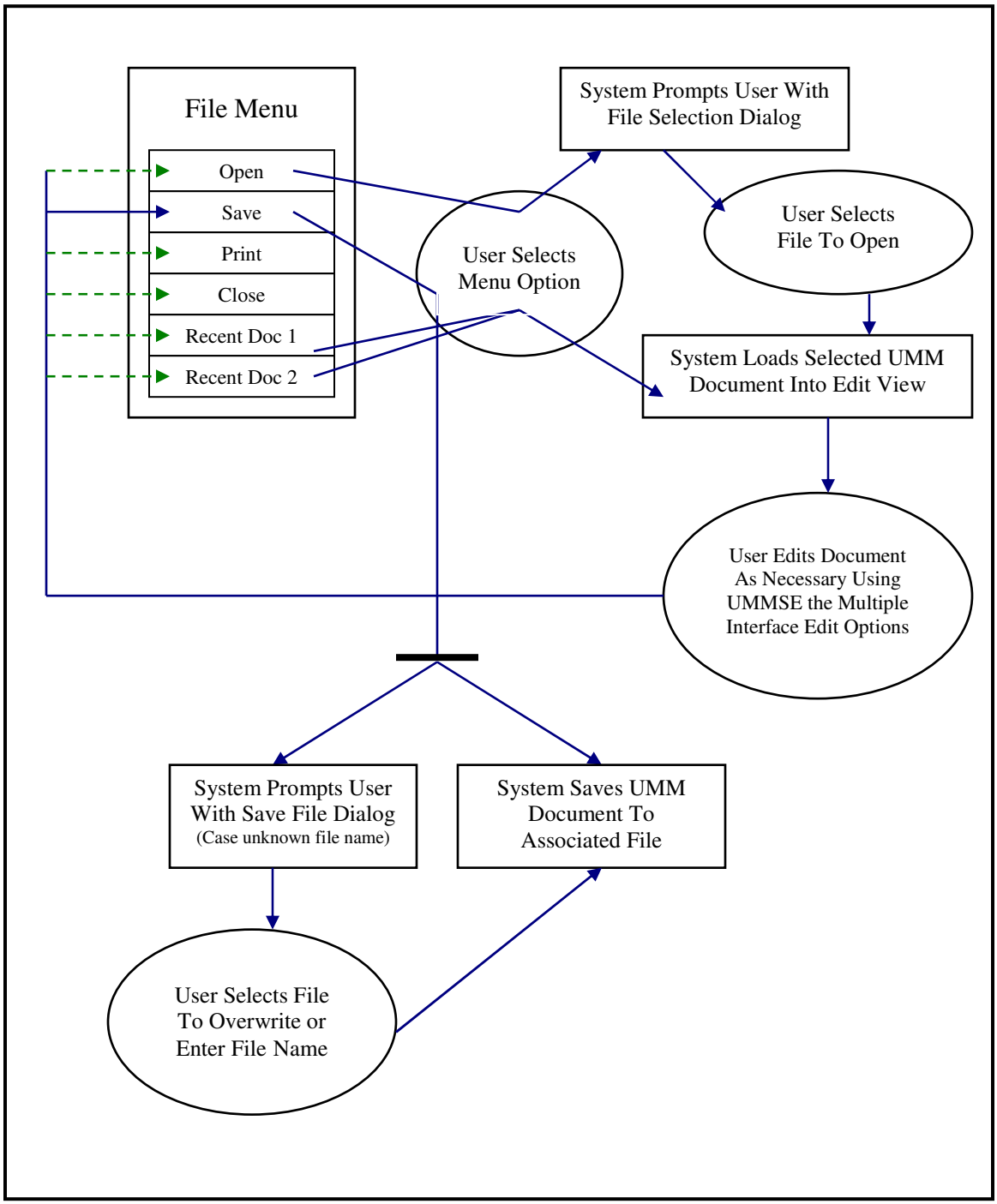


Figure 3.1. UMMSE Activity Diagram

Tasks Represented:
Start UMMSW To Create New UMM Document
Edit Current UMM Document Using UMMSW
Exit UMMSW Before Completion of New UMM Document
Progress Through UMMSW Towards Document Completion

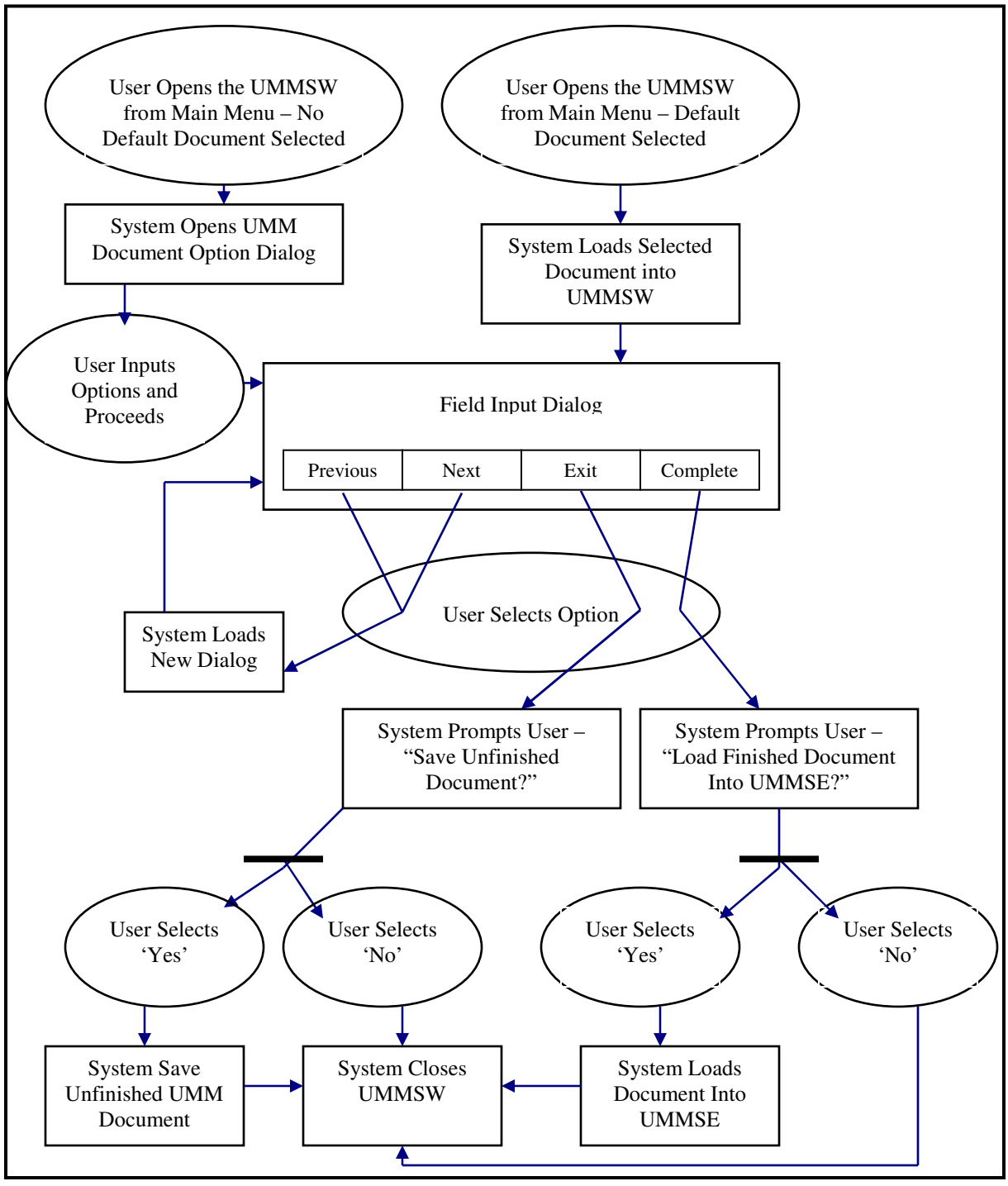


Figure 3.2. Wizard Activity Diagram

3.6 Survey and Analysis

To gather actual data concerning the UMMSE application user requirements a set of prospective UMMSE users was surveyed. The questionnaire issued to the subjects is provided in Appendix A. It focuses on two major aspects of the user application. The first aspect focuses on UniFrame specific questions. These questions are designed to extract data concerning prospective user knowledge concerning UniFrame concepts and terminology. The second aspect focuses on application specific ideals. They are designed to determine the users' familiarity with the application's environment, including Windows™ and document style interfaces.

The survey uses three styles of questions to gather information. The first style forms questions in a rating style using a scale of 1 (little) to 5 (very). This style is used to determine a user's competency concerning a particular question. The second style uses a rating style using a scale of 1 (least important) to 5 (most important). This style is used to understand a user's preference of proposed features of the application. The third implements a free form input style. Questions that use this style request that the subject provide favorite software implementation for a particular task.

The survey was administered to a select group of persons associated and familiar with the UniFrame research project. Before the users took this survey, they were provided a presentation concerning the UMMSE application proposal and its feature requirements. Throughout this discussion subjects were permitted to ask clarifying questions as well as make user and functional suggestions. The survey is intended to gather user views of the UniFrame project, windows environments, and prospective UMMSE features as provided in the presentation. Compiled results (Appendix B) and subsequent analysis are to be used to further advance the UMMSE application towards a complete functional application for implementation within the UniFrame environment. User analysis is pertinent in obtaining this goal, as the application follows a client model with full interaction capabilities with individual users.

The survey consists of three different question formats: individual rating, multiple choice rating, and free-form input. Analysis is segmented between the different forms of

question; each being analyzed using the characteristics of its intention, resulting input, and possible error. In analyzing the resulting input, a ‘what does this mean’ approach is taken in addition to a listing of possible influences this will have on the UMMSE research development.

Individual ratings provide the survey subject with a specific question and expect a rating response of 1 to 5 that reflects the subject’s opinion. The survey consists of eight questions of this format, four that reflected UniFrame ideas, and four that targeted the Windows™ operating system environment. The first two questions are intended to develop a basis of the type of users that are taking the survey. Based on the input for these two questions, the assumption can be made that this particular subject class contains a high knowledge base of the UniFrame project and the concept of a UMM. This high rating provides some validity to user responses of the subsequent two questions, which emphasize two important feature sets (in general terms) of the UMMSE application. It is important that these two questions result in high ratings as they encapsulate the driving force behind the development of this application. Survey input to the two questions rated a very high average of 4.5 indicating that the research project does have its place within the UniFrame system. Specifically, the questions indicate that users of the UniFrame distributed system do see the importance of a ‘smart’ application to edit UMM specification files providing a contextual help system.

The second set of rating questions targets the user familiarity with the Windows™ operating system. It is intended that the UMMSE will initially be developed for the Windows™ operating system. Therefore, these questions intend to develop a sense of a user familiarity with the OS and to determine what types of OS conventions may be important to the usefulness of the application interaction. The result of the familiarity questions (average of 4.0) indicates that the users are familiar with Windows™. Additionally, users are just as familiar with the operating system’s application shortcut conventions, a usability feature that is important to advanced users of an application. Interestingly, comments that were occasionally noted on the survey pages reflect that the surveyed users also preferred other operating systems and their features. Therefore, it can be assumed that many of the users of the UniFrame system are multi-operating system

savvy and may prefer to have a choice platform on which to build a UMM specification, a consideration identified throughout continued development of the UMMSE and its integration into the UniFrame system.

The multiple-choice ratings provided additional leeway to the survey subjects in obtaining their input. The questions were designed to rank 5 different related concepts amongst each other taking into account the importance of one over the other. This form of question introduced some confusion to the survey takers. Roughly one half of the survey takers rated each concept on a scale of 1 to 5 in importance individually, rather than the intended procedure of only using each ranking once; an obvious oversight in the question's directions. However, taking this input factor into account it is still possible to obtain useful individual results by computing the average ranking for each individual concept and then comparing the averages across all five group concepts. This method was used to obtain the following analysis concerning UniFrame and Application features.

The first five concepts concern the UniFrame project and evaluate possible features that may fill a particular functional void within the project. Using these results, it is possible to develop a feature development timeline by developing the most important features (as ranked by the survey) before those features that were not viewed in the same light. Likewise, the results also provide input determining if certain features should even be developed. It could be argued that any feature ranked below the average input (27 – or the average rank, 3, multiplied by the number of takers, 9) would not provide enough benefit to the users to warrant development within the UMMSE. The value of 27 will be used to represent this decision point. Among the five UniFrame related concepts there were no candidates for this distinction indicating that all five concepts are important to the application. Inherent version control ranked the highest among the features. The developed version scheme for UMMSE specifications addresses this ideal defined in Section 4.3. Following in ranking is the ability to edit UMM specifications in a free-form format, which is another of the three main user interfaces and features of the UMMSE. These results further justify that the UMMSE application and its features as proposed and defined are important to the UniFrame system and its users. Included in the five concepts are two concerning the importance and integration of the help topics within the

application. It has already been defined that help availability is important to the UMMSE; however, it is still unclear as to how this help should specifically be implemented. The intention of the inclusion of the two help concepts was to obtain a judgment as to which the user would prefer. The survey results indicate that users would prefer the keyword index format when compared to a contextual link format by a close margin of two rating points.

The second set of multiple-choice ratings target graphical and control concepts. These survey results will not be analyzed to the degree of the other topics as they reference the 'frills and thrills' portion of the user interface rather the functional contexts. Among the graphical concepts, an emphasis was put on those that have a more prominent role with the visual presentation of the application, including smooth scrolling and rich edit text formats. Conceptually this makes sense, as a user will be subjected to graphical features such as these more often than a feature such as a fading menu display which was ranked last of the five concepts with a value of 21. Obtaining a ranking below the defined average indicates that fading menus should be considered for removal from the implementation list. The second list of concepts to be ranked concerned application specific control features. Among the five contenders, two of them commanded much higher rankings than any of the other three contenders. The results indicate that keyboard shortcuts and file drag and drop functionality are high among the control features that users would like to have available within an application. These features outranked features like dock-able and hide-able toolbars, as well as multiple edit views, all of which ranked either at or below the derived decision point of 27 ranking points. The users' input, provided by the graphical and control concepts, provides important information that is used during the design of graphical user interfaces.

The final portion of the user survey requests input using an open-ended format and concerns the types of applications users prefer to complete particular tasks. In hindsight, this section would have been better served if the questions were not inclusive of a particular operating system, such as Windows™, which was used by the survey. This would have eliminated the conflicting view of which operating system a user may prefer, as well as broaden the input realm to include popular applications of multiple operating

systems that could have been used for feature comparison purposes. Both questions resulted in a dominant preferred application, which will be valuable when accessing the look and feel of the UMMSE. The first question targeted file document editors. This question was included to address the main concept of the application that is the ability to edit a document style file. The primary response to this question is Microsoft Word™ document editor. The second question addressed development environments and was included for two reasons. First, it is expected that the initial UMMSE users will be familiar with programming, and therefore, would be familiar and comfortable with programming environments. Secondly, the definition of a UMM specification is similar to a development environment in that it relates technical aspects of a developed component. Among the provided user responses, Microsoft's visual development environments accounted for the majority of input (citing both Visual Studio C++ and Visual Studio .NET).

In conclusion, the survey resulted in excellent input from prospective users of the UMMSE application. The results did indicate some implementation error, both in the forming of questions, as well as the underlying theory of the questions. For example, the survey was geared towards Windows™ users (guided by the expected UMMSE environment). However, this inherently excluded users from providing OS independent information that could have provided unique insight to the development of UMMSE. The survey also provided a base of interpretation and assessment of validity concerning many of the concepts and features that had been documented during the user development and feature analysis phases. Overall, the survey served its purpose in implementation, developing new angles of viewpoint to be accessed, encapsulating a varying amount of important user information provided by a small set of specially influenced users of the UniFrame system.

4. FUNCTIONAL SPECIFICATION

This chapter presents the functional specification phase implemented in the UMMSE development process. The specification defines major functionalities of the application and how they interact with the two major graphical user interfaces. Section 4.1 details the purpose of the specification within the domain of the software development project. Section 4.2 develops the functional specification data including the application GUI models and feature requirements. The UMMSE application is highly dependant upon the UMM specification format and its data. Formalization of the UMM specification, definition of its researched state, explanation of imposed modifications, and its impact on the application integration is addressed in section 4.3.

4.1 Specification Phase

The specification phase develops user and usability requirements by formally specifying major functionalities to be implemented. In organizational projects, the intent of this step in the development process is to merge the conceptual ideals of the application offered by the creation and analysis teams, with the technical qualities of the engineering team. In the domain of this project, the specification's breadth and formality is not as stringent as a project concentrated on the specification would incur. The emphasis of the functionalities provided in this specification is instead displaced in favor of the design phase.

Two major functional concepts are provided by this specification, 1) specific tasks that the software will enable for the user, and 2) the actions of the user to perform the tasks [ETZ00, 5]. The tasks specified in this specification encapsulate the major features of the application; more subtle tasks are left for clarification during the design phase. Actions of the user required to perform a particular are specified in the interface

implications portion of the specification. Also included are the possible system responses that complete the user interaction process.

Though natural language is used for the specification, it is of a more formalized nature than that used for conceptualization (usability). The formalization process may also include pictorial representation of task accomplishment by using relationship diagrams and flow charts [GSA00, 10-7]. The importance of this process to UMMSE is the relationship between the two major user interface modules, the editor and wizard, and their associated features. Functional to module mapping diagrams are provided to emphasize the interaction points between each module and its features. Conveniently, this provisioning serves two purposes. It provides additional documentation support for the design and implementation of the application's GUIs. Additionally, the paths defined within the diagrams provide a verification checkpoint of proper flow control during the implementation and operation phases of the waterfall development model.

4.2 Application Modules

There are two major high-level modules that have interface implications within the UMMSE project. Each of these modules encapsulates major UMMSE features, user-interface implications, and must interface with other back-end modules. This section of the feature development document provides a brief description of each module and its place within the UMMSE structure. Following this section, feature requirements (4.2.1) and module to functionality mappings (4.2.2) are detailed to provide a more in-depth view of the feature development.

1) Application Menu \ Document View:

This is UMMSE's main view and includes the application's main editing, menu, and toolbar views. The user-interface presentation is a standard Windows™ document view. When the application is loaded, this view is the first view presented to the user. From this view, it is expected that the user should be able to

access all functionalities of the application when appropriate for execution. Documents (UMM specifications) are edited in this view using the document window, which is the main window of the interface.

2) Unified Meta-Component Model Specification Wizard (UMMSW):

One of the main goals of the UMMSE application is to provide the ability to create proper UMM specifications in a timely manner. To do this, a quality user-interface must be designed that allows data error checking, flexible input, and intuitive user prompts. UMMSW features are designed to accomplish these main application goals. The wizard is a secondary form of the document view that integrates the UMM specification editing process with a more user-informed dialog interface. There are two methods of entrance to the module, each of which serves a unique purpose. This first encapsulates the entire process of creating a UMM specification and uses the complete wizard layout from start to finish. The second allows a user to open the wizard at a particular input element in context to the user's current position in the document edit view. Each of these access points is detailed by the feature requirements.

4.2.1 Feature Requirements

Feature requirements define the features that must be included in the design of the UMMSE application. The requirements are partially derived from performed usability analysis (sections 3.2 and 3.3). Each requirements lists the attributes' name, module, purpose, interface implications, and future support. Each attribute has a specific purpose within the feature requirement, defined in the following format:

Module: Implementing Module
Purpose: Summary of purpose within UMMSE; goal of feature; current unknowns.
Interface Implications: Brief description of user interaction, pre-conditions and post-conditions of interactions.

Future Support (optional): Offers related features that would provide increased functionality though are beyond the scope of this research project.

4.2.1.1 Open UMM Document

Module: Application Menu \ Document View
Purpose: Opens a UMM XML format specification into the document view. This is an entry feature of the UMMSE that provides paths to multiple editing functions. It should accurately display a UMM specification to the user in a mode that implies editing functionality. This is a key feature of the UMMSE and should focus on accurately displaying UMM specifications and usability.
Interface Implications: This feature is offered through three different user interactions, 1) menu option, 2) tool icon, and 3) a keyboard shortcut. It must provide all common editing functionality as well as a quick access to the context help features. Before the use of this feature, the UMMSE should be in an idle state. The idle state is defined as a state where no current user processing is occurring on any other interfaced features. It should be possible to have multiple documents open at the same time so the UMMSE may have other documents open while in an idle state. After opening a UMM document all common document features (such as close and save) are to be available. Following the closure of this feature the UMMSE returns to an idle state.

4.2.1.2 Save UMM Document

Module: Application Menu \ Document View
Purpose: Saves a currently opened UMM specification document to its defined XML format. This is one of the key purposes of the UMMSE in that it provides the ability to permanently save modified specifications.
Interface Implications: Feature is offered through three different user interactions, 1) menu option, 2) tool icon, and 3) a keyboard shortcut. These interactions will only be displayed as ‘available’ once a document has been opened into the UMMSE and at least one modification has been made to the document. The availability of the ‘save as’ variation does not require a modification to be made. Following the execution of this feature, the UMM specification will be saved to the selected file path in its defined XML format. A variation of the feature, ‘save as’, provides the ability to save a document to a new file path as opposed to that currently set for the document.

4.2.1.3 Edit UMM Document

Module: Application Menu \ Document View
Purpose: The ability to edit UMM specifications in a controlled manner, external to its defined structure, is the main goal of the UMMSE. Once a UMM document has been opened it should be possible to perform multiple editing functions, all of which will be listed under the edit feature. There are two possible input methods for document editing, 1) keyboard input and 2) mouse input.
Interface Implications: Editing features are made available for any open and unlocked UMM specification by the UMMSE document view. The features are to follow standard Windows™ conventions to provide external consistency. Keyboard Tab control allows a user to navigate the input fields sequentially.

4.2.1.4 Toggle View

Module: Application Menu \ Document View
Purpose: The UMMSE provides the ability to have multiple UMM specification documents open for editing at any particular moment. This ability introduces the need for the user to easily toggle between the opened documents, a feature that has become common within multiple document interface (MDI) applications.
Interface Implications: There should be two different methods of accessing the document toggle feature. The first method is provided by combination keyboard shortcut that iterates the toggle view through the open document list in sequential order. The second method uses a check-marked 'open' document list that can be accessed from the associated toolbar button. This button provides a drop-down menu that lists all currently opened UMM documents and from which the user can choose to toggle into view; the currently selected document is denoted with a check mark beside it. Following selection of the document to be toggled into view, the document view window is populated with the information of the selected UMM specification in a maximized format.
Future Support: A third method of document toggle (open selection) access could be provided by the use of a 'desktop' view within the document view window. When all open documents are in a minimized state the document view window will consist of desktop view that has an icon (with document name text) of each opened document. At this point a user is capable of maximizing or closing any document in view.

4.2.1.5 Access UMMSW

Module: Application Menu \ Document View
Purpose: The menu and document view should offer two methods for accessing the UMMSW module. The first access point is from a standard menu and tool bar option. This option opens the wizard into either a blank view or into a view of a loaded UMM specification. The second access point is generated from 'quick reference' wizard points that can be accessed by a context specific pop-up menu from each editing feature.
Interface Implications: As this feature's purpose notes, there are two different variants of this feature that have interface implications. The wizard should be able to be opened at anytime into a blank state. This state starts the wizard at its opening presentation screen that prompts for initial file and UMM specification attributes required for the creation of a UMM. The second access point provides the ability to load a currently opened UMM document into the wizard format. This load should be performed in a context environment, meaning that it should be opened into a screen relative to the field that generated the open command. The wizard provides a more structured editing environment than the document view editing features; it also provides additional guidance and directions, for these reasons it is logical to have context access points available from every edit field in the document view.

4.2.1.6 Access Context Help

Module: Application Menu \ Document View
Purpose: The goal of the context help integration is to provide specific and easily accessible help at any point during the process of creating a UMM specification. The realm of this feature in the context of the editor is not to create the complete help information module, but only to create a method of logical access of the help to the user. The help information module is a separate back-end module in itself and integrates with multiple modules with the UMMSE.
Interface Implications: There are two different forms of help to be offered through the interfaces of the editor. The first comes in the form of the plain text visible on every dialog and its content is to be consistent with what field of data is currently being prompted. The information provided by this form of help is limited when compared to the information provided by the context help link. Putting the application into a context wizard status using a toolbar option accesses this control mode. When clicked this link will open a HTML based information guide – more specifically the information guide will open focused on the help that is specific to the data being requested on the generating dialog.
Future Support: There are multiple opportunities to improve the user experience concerning how help information is accessed. Rather than only providing an application control mode, an icon link to access specific context help could be provided near each input field. Secondly, additional help could be offered in the form

of an on-line database link. This would link would open an indexed table of help contents located on a network. The initial ‘index’ would be related to the generating entry field.

4.2.1.7 Data Entry

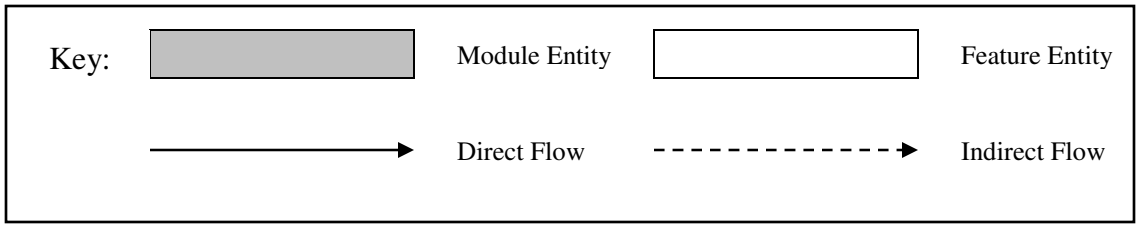
Module: UMMSW
Purpose: The main goal of the UMMSW is to provide specific UMM specification data entry fields with specific direction. To accomplish this input, multiple dialogs with multiple data entry fields per dialog are used. The purpose of the fields is to provide an error restricting form of input without limiting the user’s flexibility to create a tailored UMM specification. The activity of ‘data entry’ is a generalization of a feature that encompasses a large part of the editor interface’s functionality.
Interface Implications: As noted by this feature’s purpose, there are multiple options, selections, and data fields that pertain to this high-level feature. It is not the purpose of this section to breakdown this information though it is important to note that consistency, coherency, and clarity are major factors concerning the success of the UMMSW. The data entry feature is accountable to the majority of the user interfaces implicated with the wizard. Independent of what fields are required by the UMM specification format, all data entry of similarity should use the same visual and control schemas.

4.2.1.8 Exit \ Save \ Finalize Wizard

Module: UMMSW
Purpose: This feature serves the purpose of either saving or finalizing the UMMSW specification creation process. Depending on the hosting dialog of the feature, it will either provide the ability for a user to save current incomplete work or finish creating a complete UMM specification.
Interface Implications: This option should be made available on every dialog (except the initial options dialog) throughout the UMMSW. Upon selecting the action of this feature, which will be offered in the form of an interface button, a selectable mini-dialog will be provided. This dialog offers the possible courses of action available, which is dependant on the completion status of the UMMSW process. There are three possible actions that could be made available: exit, save, or complete (create). Upon selection of a choice from this dialog the UMMSW will continue to execute according to the action selected.

4.2.2 Mappings

Mapping diagrams provide a visual representation of functional features to module relationships. Each diagram has two entity components and two flow control attributes. The first entity is the module representation that is the centralized module being described (document view or wizard view). The second component is a set of entities that represents the features with which the module is interacting. The first flow control indicates that a user directly invokes the functionality. The second flow control represents an indirect return path that the application may take inherently from a user action. The following diagram syntax is used to depict the relationships:



4.2.2.1 Document View

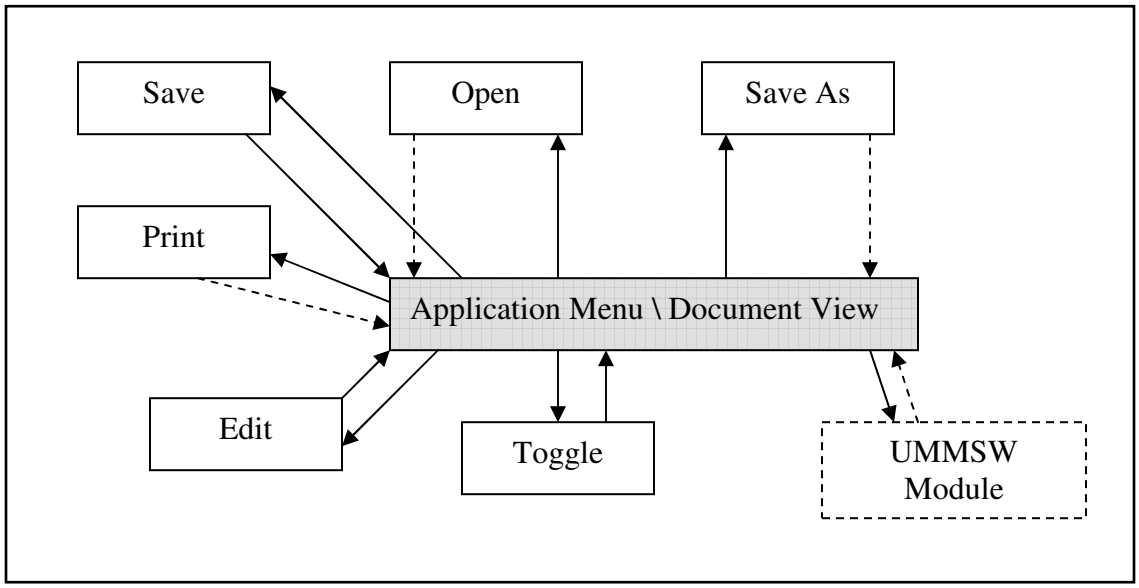


Figure 4.1. Document View Module Mappings

4.2.2.2 Wizard View

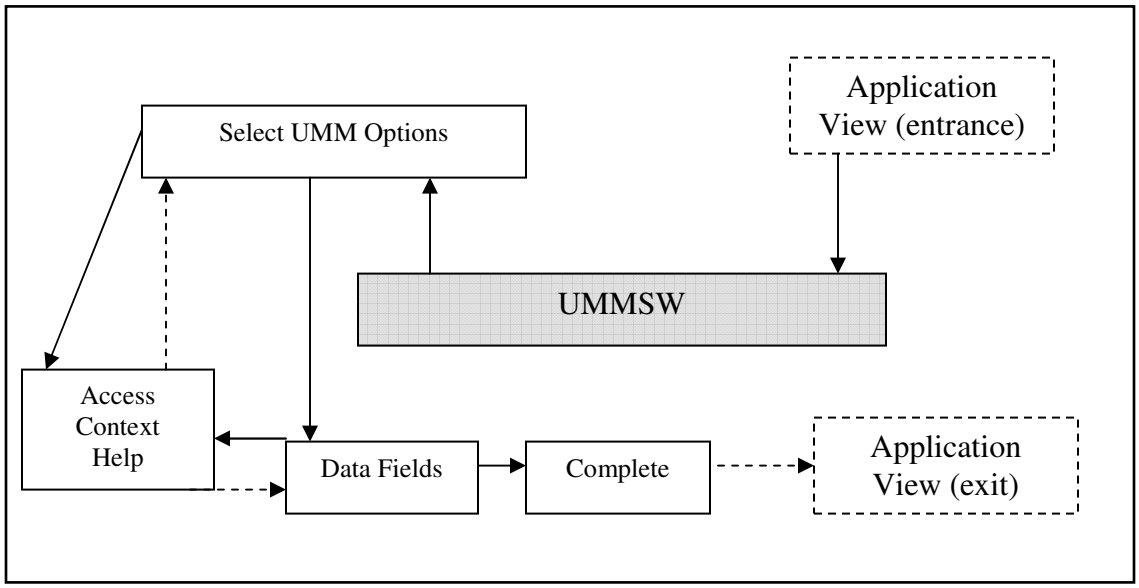


Figure 4.2. Wizard View Module Mappings

4.3 UMM Specification

A goal of the UMMSE is to open up the UMM XML formatted specification to the user through a document edit interface. To do this, a successful transition from the standard UMM specification XML document into an editable UMMSE interface format is required. This process is dependent upon certain knowledge of the UMM specification standards, format, and content definitions. The purpose of this section is to identify this knowledge and digress on its implications on the UMMSE interface. To accomplish this, a formal definition of the UMM specification must be established. This is done in two segments. First, previous research provides a content basis that identifies the current UniFrame ideal of the specification's data content. The second segment realizes modifications required to support formalization of the specification required for third party software communication (such as UMMSE). Completion of these two processes allows the composition of the first formalized UMM specification, version 1.0.0 (4.3.3).

Lastly, integration analysis (4.3.4) is presented to address the specification's impact on UMMSE interfaces.

4.3.1 Research State

This section provides the state of the UMM specification before modifications are added to complete the formalization process. It provides a list of specification data defined by multiple sets of element names and attributes. XML uses a standardized structural format composed of element nodes and attribute tags. Element nodes may be embedded within each other to form a hierarchy of element relationships. The list specifies each element and attribute by referencing its hierarchical parents assuming that all elements are a child of the root (highest) node. Data for this compilation comes from the "Component Structure Specification" as defined in *Formal Specification of Components in UMM* [RAJ01, 2.1: 4.2.1] and *The UniFrame System-Level Generative Programming Framework* [HUA00, 25-26]. A description of each attribute can be found in Table 4.3.

Syntax: [parent 1] : […] : [parent x] : [Element] : Attribute

Name
Description
Subcase
FunctionalDescription
FunctionalDescription : Function
ComputationalAttributes
ComputationalAttributes : InherentAttributes
ComputationalAttributes : InherentAttributes : Author
ComputationalAttributes : InherentAttributes : Version
ComputationalAttributes : InherentAttributes : DateDeployed
ComputationalAttributes : InherentAttributes : ExecutionEnvironment
ComputationalAttributes : InherentAttributes : ComponentModel
ComputationalAttributes : InherentAttributes : Validity
ComputationalAttributes : InherentAttributes : Structure
ComputationalAttributes : InherentAttributes : Registrations
ComputationalAttributes : InherentAttributes : SystemName
ComputationalAttributes : InherentAttributes : DomainName
ComputationalAttributes : InherentAttributes : Validity
ComputationalAttributes : InherentAttributes : Atomicity
ComputationalAttributes : FunctionalAttributes

Table 4.1. Attribute Research (cont.)

ComputationalAttributes : FunctionalAttributes : TaskDescription
ComputationalAttributes : FunctionalAttributes : AlgorithmAndComplexity
ComputationalAttributes : FunctionalAttributes : Alternatives
ComputationalAttributes : FunctionalAttributes : Resources
ComputationalAttributes : FunctionalAttributes : Resources : Architecture
ComputationalAttributes : FunctionalAttributes : Resources : Speed
ComputationalAttributes : FunctionalAttributes : Resources : Load
ComputationalAttributes : FunctionalAttributes : DesignPatterns
ComputationalAttributes : FunctionalAttributes : Usages
ComputationalAttributes : FunctionalAttributes : Aliases
ComputationalAttributes : FunctionalAttributes : Functions
ComputationalAttributes : FunctionalAttributes : Functions : BehavioralContract
ComputationalAttributes : FunctionalAttributes : Functions : BehavioralContract : Precondition
ComputationalAttributes : FunctionalAttributes : Functions : BehavioralContract : Invariant
ComputationalAttributes : FunctionalAttributes : Functions : BehavioralContract : Postcondition
ComputationalAttributes : FunctionalAttributes : Functions : ConcurrancyContract
ComputationalAttributes : FunctionalAttributes : Functions : Technology
ComputationalAttributes : FunctionalAttributes : Functions : Syntactic Contract
CooperationAttriubutes
CooperationAttriubutes : ExpectedCollaborators
CooperationAttriubutes : PreprocessingCollaborators
CooperationAttriubutes : PostprocessingCollaborators
AuxiliaryAttributes
AuxiliaryAttributes : FaultTolerant
AuxiliaryAttributes : Security
AuxiliaryAttributes : Mobility
ServiceAttributes
ServiceAttributes : ExecutionRate
ServiceAttributes : ParallelismConstraint
ServiceAttributes : Priority
ServiceAttributes : Latency
ServiceAttributes : Capacity
ServiceAttributes : Availability
ServiceAttributes : OrderingConstraint
ServiceAttributes : QualityOfResultsReturned
ServiceAttributes : ResourcesAvailable
ServiceAttributes : ResourcesAvailable : Hardware
ServiceAttributes : ResourcesAvailable : Software
AuxiliaryAttributes : Mobile
AuxiliaryAttributes : ComponentSecurity
AuxiliaryAttributes : ComponentSecurity : Method
AuxiliaryAttributes : ComponentSecurity : Level

Table 4.1. Attribute Research

4.3.2 Modified State

This section advances formalization of the researched UMM specification state into version 1.0.0 of the UMM specification. This is accomplished by first defining added and modified elements and attributes provided by the project (4.3.2.1). The added and

modified attributes are then merged with the previously identified research attribute list. In addition, this process adds type information to each attribute by declaring each attribute's software data type (Section 4.3.2.2). With these two tasks accomplished, this section concludes by detailing the final formalization of the UMM Specification (Section 4.3.2.3).

4.3.2.1 Attribute Modifications

Syntax: [parent 1] : […] : [parent x] : [Element] : Attribute

Date
ChangeList
ChangeList : Change
ChangeList : Change : Editor
ChangeList : Change : Date
ChangeList : Change : Description
ComputationalAttributes : Algorithms
ComputationalAttributes : Algorithms : Algorithm
ComputationalAttributes : Algorithms : Algorithm : Name
ComputationalAttributes : Algorithms : Algorithm : Complexity
ComputationalAttributes : Algorithms : Algorithm : Description
ComputationalAttributes : Algorithms : Algorithm : Function
CooperationAttriubutes : Collaborator :
CooperationAttriubutes : Collaborator : RelatedFunction
CooperationAttriubutes : Collaborator : ExpectedCollaborator
CooperationAttriubutes : Collaborator : RequiredCollaborator
CooperationAttriubutes : Collaborator : ProvidedCollaborator
ServiceAttributes : QualityOfService
ServiceAttributes : QualityOfService : Attribute
ServiceAttributes : QualityOfService : Attribute : ID
ServiceAttributes : QualityOfService : Attribute : Value
ServiceAttributes : QualityOfService : Attribute : Extension
ServiceAttributes : AvailableResources
ServiceAttributes : AvailableResources : AdditionalReference
ServiceAttributes : AvailableResources : AvailableResource
ServiceAttributes : AvailableResources : AvailableResource : Types
ServiceAttributes : AvailableResources : AvailableResource : Data
ServiceAttributes : AvailableResources : AvailableResource : TypeUnit
ServiceAttributes : AvailableResources : AvailableResource : DateUnit
ServiceAttributes : AvailableResources : AvailableResource : ResourcePair
ServiceAttributes : AvailableResources : AvailableResource : ResourcePair : TypeValue
ServiceAttributes : AvailableResources : AvailableResource : ResourcePair : DataValue

Table 4.2. Attribute Modifications

4.3.2.2 Attribute Compilation

Table 4.3 represents a merge of attributes sourced from both the research and modified state lists. Four characteristics are provided for each attribute: Tag Id, Software Data Type, Required, and Description. Attributes are categorized under their nearest parent, referenced by the first proceeding bold parent. There are five tags that are the highest parent tags other then the root node. They are ComponentAttributes, ComputationAttributes, CooperationAttributes, AuxiliaryAttributes, and ServiceAttributes; such nodes bound parent hierarchy trace backs. Tag Ids use the formal node attribute tag that appears in the formal UMM specification XML representation. There are four possible software data types: boolean, integer, float, and string. The requirement and domain column specifies whether the field must be populated and what possible values it may be populated with (if applicable). Column four provides a brief description of the attribute in the context of the UniFrame environment.

Two additional notations are used within this table. It is possible that a parent specification (element) be permitted to have multiple occurrences within the specification to create a list of elements with different attributes. In this case the keyword 'multiple' is specified after the element heading. A point of reference is also noted. Those descriptions that are followed with an asterisk (*) have been cited from *The UniFrame System-Level Generative Programming Framework* [HUA00].

Tag Id	Data Type	Require\Domain	Description
UMMSpecification			
Name	String	Required	Specification name.
Version	String	Required	Version of UMM specification format. Defined by specification DTD.
Specification			
AttributeList			
Presentation	String	Required {abstract, concrete}	Indicates presentation of the specification.
ComponentAttributes			
Author	String		Author of specification.
Name	String		Component Name

Table 4.3. Attribute Formalization (cont.)

Tag Id	Data Type	Require\Domain	Description
Subcase	String		Indicates information related to communication patterns of functions of the components. These reflect the synchronization aspect of the functions. *
Description	String		Natural language description of the component.
ChangeList			
Change (multiple)			
Editor	String		Name (author) of this specific change.
Date	String		Date of this specific change.
Description	String		Description of the changes made to the specification for this change.
ComputationAttributes			
Inherent			
Version	String	Required	Version of the described component.
Author	String	Required	Author (Developer) of the described component.
DateDeployed	String		Deployment date of the component. This is the initial date of component availability.
ExecutionEnvironment	String		Indicates the particular environment of component execution. Typically limited to the operating system.
SystemName	String		Indicates the system family to which the component belongs. *
DomainName	String		Provides the domain scope for the component. *
ComponentModel	String	{Domain, Wrapper, Representation, Classification, User Interaction, Headhunter}	Describes the model type of the component.
Validity	Integer		Provides the time to live for described component. Time to live is calculated using the deployment date as its starting point.
Atomicity	Boolean	Required	Indicates whether the component is atomic. *
Registration (multiple)			
Data	String		Represents a particular registry at which the component is registered.
Functional			
Resources			
Resource (multiple)			
Architecture	String		Indicates the type of architecture of this particular resource. Example: PCI Bus.
Speed	String		Indicates the speed at which this

Table 4.3. Attribute Formalization (cont.)

Tag Id	Data Type	Require\Domain	Description
			resource must execute. Example: 2 MHz.
Load	String		Indicates the load that resource must be capable of handling. Example: 100 Mbps
Algorithms			
Algorithm (multiple)			
Name	String		Name of this algorithm.
Complexity	String		Complexity of this algorithm. Typically denoted in O (big-O) notation.
Description	String		Description of this algorithm. May include implementation details.
Function	String		Indicates component function that algorithm relates to. 'Global Component Function' is used to represent the entire component.
FunctionsAndContracts			
Function (multiple)			
Name	String		Natural language name of function.
Description	String		Description of purpose and design goal of this function.
Technology	String		Technology used by this function.
SyntacticContract	String		Computation signature (prototype) for this function. Uses the proper syntax of high-level programming language. *
ConcurencyContract	String		Description of this function's concurrency control; including synchronization, semaphores, counters, and monitor usage.
Precondition	String		Complete description of functional preconditions.
Postcondition	String		Complete description of functional postconditions.
Invariant	String		Complete description of dependencies that may not change and must be satisfied.
DesignPattern (multiple)			
Data	String		Indicates a design pattern employed by the component. *
KnownUsage (multiple)			
Data	String		Indicates a known usage of the component or proposed use of the component. *
Alias (multiple)			
Data	String		Indicates representative alias for the component. *
CooperationalAttributes			
CollaboratorList			

Table 4.3. Attribute Formalization (cont.)

Tag Id	Data Type	Require\Domain	Description
Collaborator (multiple)			
Function	String		Indicates component function that this collaborator relates to. 'Global Component Function' is used to represent the entire component. Only specified components functions may be used.
ExpectedCollaborator	String		Provides an unrestricted general view of expected collaborators for use of this component function.
RequiredCollaborator	String		Provides components and other entities that this component function depends on.
ProvidedCollaborator	String		Provides components and other entities that depend on this component function.
AuxiliaryAttributes			
Mobile	Boolean	Required	Indicates whether component is mobile. *
Mobility	String		Provides additional details concerning component's mobility if mobile.
FaultTolrance	String		Indicates the fault tolerance scale and level of this component.
SecurityList			
ComponentSecurity (multiple)			
Method	String		Describes a security method used by component to implement security of data storage and transmission.
Level	String		Indicates the level of security used by the component. May be represented in an arbitrary manner also described by this attribute.
ServiceAttributes			
ExecutionRate	Float		Rate of execution once invoked.
ParallelismConstraint	String	{synchronous, asynchronous}	Indicates whether component is capable of parallel execution.
OrderingConstraint	String		Describes constraints imposed when ordering this component for use.
QualityOfService			
QosAttribute (multiple)			
ID	String		Natural language text representation of the quality of service attribute being referenced.
Value	String		QOS value that corresponds to the same level id and is extensively described in by the extension attribute.
Extension	String		Provides a description of additional

Table 4.3. Attribute Formalization (cont.)

Tag Id	Data Type	Require\Domain	Description
			information that may be required to interpret the attribute's representation.
AvailableResources			
AvailableResource (multiple)			
Type	String		Entry that indicates the type of architecture (software or hardware) referred to for this resource. Example: Digital Signal Processor.
Data	String		Entry that indicates the data that corresponds to this resource. Example: Widgit Transactions
TypeUnit	String		Describes the unit of measurement or representation used to represent variants of the resource requirement. Example, Speed – MHz.
ValueUnit	String		Describes the unit of measurement or representation used to represent the value of the corresponding type variant. Example, TPS (transactions per second).
AdditionalReference	String		Provides a method of relaying further information references to the specification reader. Typically this references a public URL.
ResourcePair (multiple)			
TypeValue	String		Provides a value of this pair's type, understood to be in the unit measurement of 'TypeUnit'. Example, 133 (implied MHz). If multiple resource pairs are specified for graphing purposes then this is the x-coordinate.
DataValue	String		Provides a value of this pair's data that corresponds to 'TypeValue.' Understood to be in the unit measurement of 'ValueUnit". Example, 100 (implied TPS). If multiple resource pairs are specified for graphing purposes then this is the y-coordinate.

Table 4.3. Attribute Formalization

4.3.2.3 Formalized UMM Specification

The final step in creating the formalized UMM specification is to transform the formalized attribute table into a XML Document Type Definition (DTD). Represented as in tree structure, the DTD is one of the standardized formats used to describe a particular set of data. By using either XML DTD or XML Schema, any database of information can be processed generically to extract the document information [ELM00, 848]. Applications, such as UMMSE, need only to know a set of data's DTD in order to properly interpret the data and perform functions on the data. For example, UMMSE reads a UMM specification with a prior interpretation of the DTD. The interpretation is then applied to the data mappings to define specific interfaces. Furthermore, UMMSE writes UMM specifications following the DTD format so that other applications may use the generated data file without conflict.

A brief introduction to the major data tags used in the formal UMM specification DTD is provided. The root UMM specification tag is *UmmSpecification*. All subsequent document nodes are children of this element. *Specification* is the next child and represents a complete component specification. It would appear that this serves the same purpose as the root tag. However, the *Specification* tag may be multiple within the document to separate related but individual component specifications. This provides the possibility of including multiple component specifications in the same document. Each specification has five major attribute lists that fall under the *AttributeList* tag. There can exist only one *AttributeList* and its five child nodes per specification. Its child nodes are *ComponentAttributes*, *ComputationAttributes*, *CooperationAttributes*, *AuxiliaryAttributes*, and *ServiceAttributes*. These tags divide the hierarchy of the specification into related tree branches. Each of these branches contains many tags and attributes as noted in Table 4.3.

The complete DTD for UMM specification version 1.0.0 is available in Appendix C. An example UMM Specification that follows this formal DTD is available in Appendix D.

4.3.3 Version Control

It is expected that the definition of the UMM specification will change overtime as the UniFrame project is developed. Changes may consist of data reorganization, added and removed data, or data domain constraints. Not all of which can occur transparently to systems integrated with the UMM specification. Due to this, it is imperative that the specification be able to support version control. The UMM specification supports version control using a formatted version string. This string presents itself as an attribute of the *UmmSpecification* element. Being the root node of the specification, it is a reasonable assertion that the hierarchical location of the attribute will not change. Therefore, any processing application reading a UMM specification XML document has the ability to first read the specification's written version.

The formal syntax of the UMM specification version string is "Uframe.Umm.X.Y.Z". It consists of five meaningful elements. The first element is the global project space. Currently there is only one valid value for this field, 'Uframe' (abbreviation for UniFrame), however, when the project expands it may be necessary to subdivide this element. The second element indicates the direct referencing environment. Currently, it also only has one valid value, 'Umm', though the option remains available to create additional values when applicable. The next three elements have a greater impact on processing systems as they described the actual context of change from the previous revision. A change in the X element indicates that a major high-level hierarchy change as been made. For most applications this would not be a passive change. The Y element represents the mid-level hierarchy tags. In the context of the first UMM specification, a mid-level tag would be a child of any of the tags mentioned in 4.3.2.3. The final element, Z, represents a minor revision change. Adding or removing a hierarchy end tag, changing a node's attribute list, and modifying an attribute's domain constitutes a minor revision change. Most processing applications should not have to change in order to support a minor revision change, or, in the worst-case scenario, must only change in order to gain the benefit of the new information (maintaining backwards capability).

4.3.4 Integration

The editor and wizard application interfaces represent the content of the UMM specification. In doing so, they must be able to accurately represent each individual data attribute in its appropriate data format. This determination is dependant on the type of data required by the data structures that have been defined in section 4.3.2.2. Each type of control to be used within these interfaces is abstractly explained in this section. Following the control's generic description is a list of all specification attributes that use the described control. Identification of this mapping between an attribute and its visual representation is developed further during the interface design phase.

4.3.4.1 Unknown Free-form Text

This control provides an interface that allows complete and unlimited free form input from a textual input device (keyboard). It is applicable in two circumstances, 1) its main function is to gather input related to data structures that have no defined boundaries and the content of which cannot be predicted, 2) a secondary function is to provide an input device for element structures that have not yet been rigorously defined. This may occur in the case where the need for a particular element is expected in the future, but the true contextual nature of the information is not yet known. This list specifies specification attributes mapped as unknown free-form text representations.

Name	DeploymentDate	Level
Description	DomainName	ID
Subcase	SystemName	Value
Editor	ExecutionRate	Extension
SyntacticContract	OrderingConstraint	Type
Technology	ExpectedCollaborator	Data
Precondition	RequiredCollaborator	TypeUnit
Postcondition	ProvidedCollaborator	DataUnit
Invariant	Mobility	TypeValue
ConcurencyContract	FaultTolerance	DataValue
Complexity	Architecture	AdditionalReference
Version	Speed	

Author
DeploymentDate
Load
Method

4.3.4.2 Known Free-form Text

This control provides the benefit of flexibility just as its unknown free-form counterparts do in the above list. Though, one additional usability and experience benefit is contained within the scope of this control. In some manner, it lists inputs already known to be valid for the structural input. This known information may either come from an internally coded source, or it may come from previously added user values retrieved by the application on each execution. This list specifies specification attributes mapped as known free-form text representations.

ExecutionEnvironment
ComponentModel

4.3.4.3 Toggle

This control selects one value from a set of valid choices. The user is able to select any choice, when chosen the selection indicator is toggled from the old choice to the new choice. At anytime, this control must have one and only one choice selected. A two-option variable, such as a boolean, is best suited for this control. In the case that more than two choices are possible, this control can quickly become confusing to the user and may lose its simplicity. This list specifies specification attributes mapped as toggle representations.

Presentation
Atomicity
Mobile
ParallelismConstraint

4.3.4.4 Selector

This control provides a method of selection from a group of multiple choices. It does not impose the restrictions that come with the toggle control. The selector does not require that a selection be made, nor does it limit the user to only one choice. A selector's options are all predetermined and therefore it cannot accept unexpected values. If this situation is possible a known free-form text control is a better interface option. This list specifies specification attributes mapped as selector representations.

Function
Validity

4.3.4.5 Add List

This control combines a form of the output control with one or more input controls. Its purpose is to leverage an interface for a multi-value input element. It does this by associating a single set of input controls with one output control, often in the form of a list. The user selects a combination of inputs and then provides an indication that this pair should be added to the collective list of paired attributes. Once this association is made the output control displays the new association and the input controls are made available for another set of input. The output control, for example a list, may be selectable. In this case a selected element will populate the input controls with their associated data for further editing. This list specifies specification attributes mapped as add list representations.

FunctionsAndContracts	Algorithms	CollaboratorList
DesignPattern	KnownUsage	Alias
Resources	ComponentSecurity	QualityOfService
AvailableResource	Registrations	

4.4 Overview of Specification Phase

This chapter detailed the specification phase used in developing the UMMSE functional specifications. In the initial task of this phase, the application's user interface modules, document view and editor, were conceptualized. For each interface module, high-level required features were specified that outlined the main functions of UMMSE application. The chapter continues in developing a formal definition of the UMM Specification. In this process, previous UMM specifications are researched to create an initial attribute list. This list is then modified to include attributes added by this project. Finally, the UMM Specification is formalized to create version 1.0.0. At the completion of this three step process, each attribute is analyzed for content and is mapped to an appropriate visual control. This phase has established functional and UMM specifications to be used as a basis for the UMMSE design.

5. APPLICATION DESIGN

This chapter presents the design phase implemented in the UMMSE development process. Design defines how major functionalities described during specification are to be implemented. Section 5.1 introduces the design approach used in the development project providing factors particular to UMMSE design. Section 5.2 discusses the concept of a multiple document interface and its impact as the application's core design. Section 5.3 presents design factors and analysis for the editor and wizard application views. The third major design issue concerns the organization of specification data, which is discussed in section 5.5.

5.1 Introduction to Design

There are two co-existing views of UMMSE application and its role as an interaction program within the UniFrame system. The UMMSE can be reviewed as a stand-alone application by eliminating the impact of UniFrame point-of-view. From the view of a stand-alone application, there must exist a complete visual representation of ideas and control accessibility of the functional core. This implies a development concentration on view designs, frame layouts, and overall content coherency.

The second viewpoint builds upon the integration points of the UniFrame system. Its integration points refer to the ideals of the UniFrame system that must be analyzed for impact on the application. UMMSE must have a functional purpose and a basis of user requirements to drive the development of the project. The purpose of this introduction is to bridge the two opposing viewpoints and develop which individual points should be used in summing the most productive whole. This section will discuss the theory and reasoning leading to the presented editor, wizard, and data designs.

5.1.1 UniFrame Functional Integration

It is appropriate to discuss the second viewpoint as the prominent factor to begin this analysis. As the usability requirements (3.3) suggest, UMMSE is to provide an efficient and user-friendly environment to perform all UMM related functionalities that a user of the system may require. Of the three elements efficiency, functionality, and functional access, efficiency and functionality incorporate the greatest integration factors from the view of the UniFrame system. These factors dictate concept points that must be evaluated during the integration process. Efficiency refers to two different aspects of the application. First, it refers to the time required to complete any task to a particular degree of qualitative measurement. Secondly, efficiency encapsulates the long-term learning curve of a particular user that affects the distribution and use ratio of the application over the global system. Prospective users will isolate an application that does not perform useful functions. This aspect will be analyzed from the viewpoint of a function's interface accessibility and user's interaction.

To discuss the first topic of efficiency, a ratio of completion time to task effectiveness is used. Using this ratio the factors' balance of concentration between one and the other can be visualized. There are two different units of measurement used in the following illustration. First, a unit of time must be defined; hours are used to measure a worker's effort put forth on a task. The second measurement is used to measure the quality of a task's final product on a scale of comparison with expected final products, referred to as points. To simplify the comparisons, each is measured on a scale of eight units (8 hours – one workday; 8 points – best product).

Using arbitrary unit values, consider an application that performs a particular method better than any comparable method in the field. Also, assume that the worker is under the pressure of a deadline. Now consider that it took a complete workday (8 hours) to obtain the two quality points. In comparison, another method can complete the same task in a quarter of a worker's day (2 hours). In this case, the ratios would be 4 (8/2) and 1 (2/2) respectively. A review of the opposite theory will provide similar results. Consider a situation where the worker is asked to complete a particular task at a quality

level of his discretion, but it must be completed by the end of the day (4 hours remain). Using an extensive method the worker completes the task at a point level of 6. Using a second method the worker could complete the task at a point level of 4, generating ratios of 2/3 (4/6) and 1 (4/4) respectively. From these examples it can logically be derived that a decreasing completion to qualitative point ratio provides the optimal choice of task performance. In the first example, it would be more logical for the worker to complete the same task in a shorter period of time; likewise, in the second example it would be more logical for the worker to achieve the better result in the same time period. Solid application design will optimize this ratio in every scenario.

A second form of efficiency that pertains to the application's UniFrame integration is its learnability characteristic. The ability for a user of any level to adapt to a new application and become an efficient user of the technology is referred to as a user's learning curve. The user analysis identified three types of users: basic, intermediate, and expert (3.1). It is apparent that it is not possible to equalize the learning curve between the three types. For a user, this means that to obtain an expert status, more must be learned than to achieve just basic status. For this reason, design of the UMMSE adheres to one basic principle when it comes to learnability. This principle implies the application goal of maximizing development effort to minimize user effort, specifically referring the user's effort towards learning the application.

In addition to minimizing the learning curve, development efforts should be made to stabilize the curve. A user should be able to span all three user levels without hitting the proverbial 'brick wall'. Implicit (leading indicators) and explicit (help documents) design methods provide consistency throughout the learning process by not targeting any particular user level. This allows a user to transit from a starting point to a basic level, basic to intermediate, and eventually to an expert level without being overwhelmed at any particular point of the process. Combining these two developmental aspects creates the ideal learning curve for this application's integration with the UniFrame system. Ideally, the learning curve should be composed of two algebraic traits; it should be linear to account for stability, and its slope should be minimal to represent a minimized user effort.

If a curve with these traits is established using the designed outlined in this chapter, then the application's learnability may be considered maximized.

The third abstract UniFrame integration factor concentrates on the importance of the application's functional access. This factor bridges the abstraction of the UniFrame integration and the stand-alone application (5.1.2). It is important to abstract controls, functionalities, and interactions to a point where is makes sense to a user familiar with the UniFrame technology and implementation. In accomplishing this, two goals must be maintained throughout the development process. First, the internal consistency must maintain the relationship between controls and UniFrame specific functions that a knowledgeable user would expect to encounter. Secondly, the external consistency with the UniFrame environment must be maintained in order to harness the power of any previous knowledge a particular user may have. These two factors also have implications on the interface integration from the viewpoint of the stand-alone application. Control and functions within the application must be maintained throughout the multi-view interfaces. This means that a particular function (UniFrame dependant or non-dependant) should be accessed consistently throughout the application. Additionally, that function should present itself to the user in a manner consistent with that associated with any other functional access point. The core of the UMMSE application is functional; therefore it is imperative that functional access points be developed in a manner that provides the most benefit to the individual user.

5.1.2 Stand-Alone Application

The second viewpoint, to be explored in an abstract manner, is that of the development of a stand-alone application. Interface integration factors that are expressed by this viewpoint in theory should be applicable to any stand-alone application that supports GUIs. In a sense, factors such as these are generic in representation, referred to as 'user-friendly' aspects, until their actual incorporation into the development environment. During the requirement and specification phases, these factors are addressed in general terms by expressing the factor as a goal of interface integration. In

the design phase, a specific display or control feature can be justified by referring to one of these generic integration factors.

Three integration factors are explored in this section. The first references to the appearance of graphical controls and direction that the user experiences. Inherently, each interface factor that provides a user a form of direction should provide it in an implicit manner. The user, without any external dependency, should know that particular factor's intent and be provided with direction. There are many ways to achieve this goal, not all of which will be addressed. However, once a method of achievement has been used, that factor should be used consistently within the entire application. Consistency is the second factor concerning the stand-alone interface integration viewpoint. The concept of consistency can be applied to about every point of interface integration; in the context of a stand-alone application the form of consistency emphasized the most is internal.

Creating consistent interfaces throughout the application benefits many aspects of the application's success including learnability and usability. The consistency factor can be applied to the third factor to be explored, overall visual design. A visual design can include multiple view types including menus, frames, dialogs, pop-ups, and many others. Independent of which type of view may be used, it is important that they are used consistently throughout the application. Additionally, it is important that they are externally consistent with the operating system's conventions, or at a minimum do not conflict with the conventions. In the early phase of development, it is sufficient to state that visual design constitutes to a large degree the effectiveness of an application and the overall experience a user will have with the application. A contributing factor of this design is the visual (visible) language. Consisting of characteristics such as color, harmony, contrast and scale, the visual language communicates information to the user using nonverbal screen graphics [HUM00, 321].

5.2 Multiple Document Interface

The UMMSE application design uses a document view architecture. It opens, creates, and edits proprietary file formats stored using structured XML. There are two

generally accepted styles of the document view architecture: single document interface (SDI) and multiple document interface (MDI). Each style adheres to standard file view and maintenance attributes that classify them as document view architectures. As the nomenclature indicates, the main difference between the two styles is the number of concurrent document views that they support. The Microsoft Developer Network (MSDN) defines a MDI in the terms of an application that allows a user to work with more than one document at the same time [MDI00]. Additionally, the style permits document inter-working without definitively ending the previously opened document. Within the SDI style document architecture a user may also work with multiple documents, but the user is required to explicitly end a previous file session before initiating a new session.

Both styles have been researched to identify the advantages and disadvantages of each. Specifically, the analysis concentrates on the MDI style, developing the reasons that it was selected as the style for the UMMSE. Furthermore, UMMSE's implementation of the MDI style architecture is a multi-view interpretation of MDI. This means, in addition to allowing the user access to multiple documents at the same time, each document may also have multiple concurrently accessible views. This combinational format is used to facilitate parallel edit and XML views as specified in UMMSE's functional requirements. In addition to the advantageous point analysis of MDI, this section concludes by justifying the use of the MDI style within this research application. The decision must be justified within two different contexts of the application. As this research application is directly related to the UniFrame research project it is imperative that the MDI style be justified from the viewpoint of the UniFrame system user. Secondly, the decision of view style must be advantageous to the implementation environment and expected visual representation.

5.2.1 Architecture Analysis

In the environment of a MDI style there are two pertinent reasons for analyzing disadvantages of the style. One is to understand possible reasons as to why a MDI

approach may not be the best solution to the UMMSE design problems. If it can be determined that the disadvantages outweigh the advantages, then it may be proposed to approach the problem from a different viewpoint. Secondly, a concise understanding of disadvantageous issues can lead to a corrective approach to the design. User issues are often introduced by software complexity that is in turn often a result of offered flexibility. Therefore, by understanding the disadvantages and deriving interaction methods to correct them, an application gains the benefit of the increased flexibility with minimal drawbacks. This resembles the tradeoff paradigm that exists in many aspects of the software development process.

One method of determining and analyzing disadvantageous user impacting issues is to study MDI application help tutorials. By analyzing the key content of such informational references, perceived problem areas can be noted by their emphasis within a particular application's review or documentation. To understand potential MDI issues, a user help website, *www.dummies.com*, offers some specific aid concerning one of Microsoft's popular MDI applications, Word 2003 [DUM00]. The site includes user tips that emphasize solutions to two key issues common with MDI style applications.

This user help article mentions two methods for switching between documents (Alt+Tab key combination and the Window menu), and indicates that the 'auto-arrange' feature may be used to tile the document windows. Each of these user controls provides a method of addressing a problem, in this report referred to as the "lost document problem". Since an MDI allows multiple documents to be viewed in a maximized state within the owning application screen space, it is reasonably understood that the other documents will not have any identifying display within the user's view. This causes the user to lose perspective of which documents are open and in which order they are layered upon each other. There are multiple solutions that can be implemented to address the lost document problem, two of which were implemented by the Microsoft Word™ application. In designing the editor view requirements for the UMMSE application additional solutions to this issue will be proposed.

The referenced article further indicates a method that users can implement to maintain their data presence on the application screen. One method is to use what the

author refers to as a “the old split-screen trick”. This method allows a user to view two different data points at the same time in less than maximized windows; additionally these data points may reside in two different documents. This method addresses the “lost data problem”. The problem occurs when a user, either by application design or inadvertent keystrokes, loses the current data position of one document when referencing other documents. The split-screen approach resolves this issue between multiple documents because the current data position can be maintained within view and within the multiple documents. However, this solution is limited because the screen’s dimensions still limit the number of concurrent document views. The main solution to this issue, which is inherent in many application implementations, is for the software to maintain a current position status per open document, including those not currently in view. When the user returns to that document using a method solving the lost document problem, the current data presence point for that particular document is explicitly noted on the screen, typically using a blinking cursor or highlighted control. This method, in combination with the split-screen method, is implemented to address this issue within the UMMSE application.

There are many advantages of a MDI over a SDI application, particularly when using a document type such as UMM Specification XML representation. Most notable is the ability to create, view, and modify multiple documents from the same application interface. Figure 5.1 provides a pictorial of how multiple documents appear in a single UMMSE interface versus a SDI implementation. Ironically, this is the same feature that introduces the complexity leading to those issues discussed when analyzing MDI disadvantages. This eliminates the need for the user to create multiple instances of an executing program on the same machine, thusly simplifying multitasking. Instead, the user task of launching multiple application instances is replaced by the user task of loading multiple application documents. This tradeoff tends to be acceptable in practice because an application has the ability to offer convenient mechanisms to accomplish this task. For example, a selectable recent document list is often provided in the visual environment to provide an easy load access to documents the user is most likely to request for viewing. An operating system is capable of providing similar features:

however it does not offer the granularity that an application can due to the global system nature that an operating system entails.

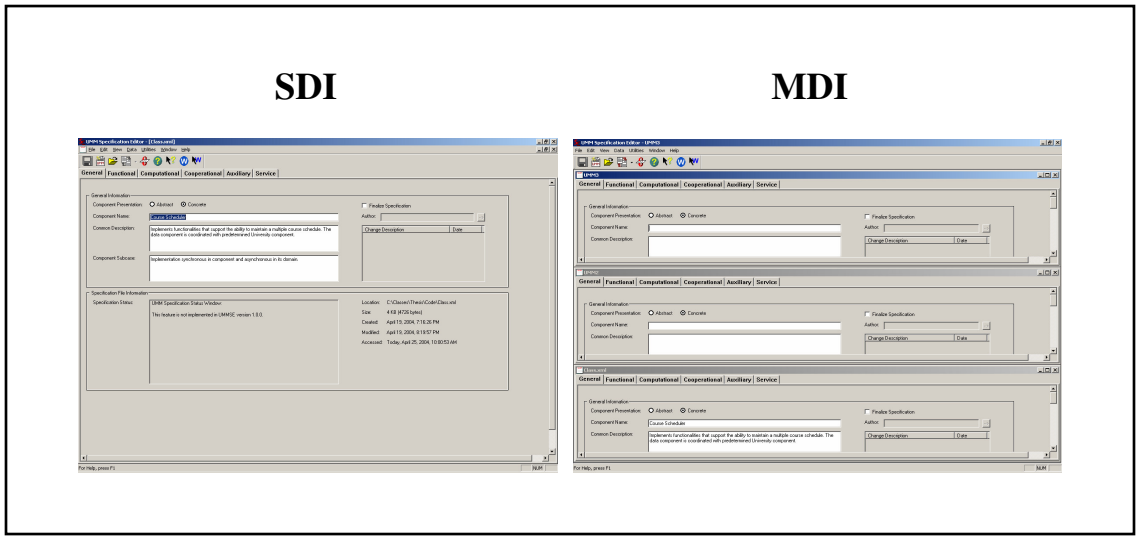


Figure 5.1. SDI Versus MDI

These described features offered by MDI style applications are designed to limit response time, provide flexibility, and simplify tasks (though it has been noted that in turn alternative complexity is added); as well, to this point they differentiate the handling of multiple documents between MDI and SDI applications. A second set of major feature controls provided by MDI over SDI reflects actual document viewing and modification. Previously described controls, split-screen viewing and shortcut document navigation provide the user inter-document capability from within the same application screen space. Again, these features may be provided by the operating system, but the implementation must be generic because it is true for all document types. In comparison, an intra-application implementation is capable of customizing the view per document type while maximizing the use of screen real estate. Similarly, data control and exchange can be customized using an MDI intra-application design. The impact of this capability has been minimized since many operating systems have implemented inter-process data communication schemes to serve this purpose. For example, Windows™ provides a clipboard that is capable of storing and retrieving temporary data blocks for transfer

between other processes. When combining a data transfer mechanism (application or operating system implemented) with one or more intra-process multi-document view schemes, the user data control is simplified by minimizing input (keyboard and pointer control) and maximizing content space screen utilization.

5.2.2 MDI Justification

The previous section has identified many advantages and feature enhancements that an MDI design can bring to an application. Using multiple techniques, introduced complexity can be minimized in attempt to make their issues transparent to the user. As initially stated the UMMSE application implements MDI. In addition to the aforementioned reasons for using a MDI system, the choice must be justified from the viewpoint of the UniFrame system and its user. Though it is expected that users will range from basic to expert, it is also expected that many users will reach the expert plateau. Combined with the expectation that a majority of users will be familiar with computer systems and applications, the added complexity of MDI is a fair tradeoff for the provided extended feature support. Another major decision factor is that the UMMSE feature specification states that each document in itself requires two possible views, standard edit and XML. This requirement lends itself to the MDI architecture. From a user standpoint, the multi-type view is still just a single document representation. However, from an architectural standpoint, each type of document view is actually a different document that just happens to have the same data content (as the underneath document is the same). In addition to the editor's MDI implementation, UMMSE offers a secondary editing environment that is based upon a SDI style. This idea is explored further when analyzing the wizard design (5.3.2). The decision to use a MDI style is further detailed by exploring the purpose and implementation of the editor view.

5.3 Interface Views

The previous section discussed the application style used by the UMMSE. Specifically, it noted that the application is structured as a multi-view and multi-document interaction tool. This section extends this information to include specifics concerning the two main user interaction view formats used by UMMSE. In explanation of the two formats, menu and dialog, their data and control access points are derived, visual restrictions stated, and context help integrations are noted. The application of the MDI architecture is expanded upon to include what effects it has on the dialog interaction style.

5.3.1 Editor Application View

The design of the editor interface requires the analysis and integration of many interface ideas. The design starts by developing methods of access to the data elements of the specification and controls of the UMMSE features. Next, the design of the editor view's interoperation with the UMMSE help controls is detailed. This section proceeds to discuss the design of the editor within the context of a MDI format, further extrapolating on the ideals of section 5.2. There is a history of GUI application integrations that specifies the trials and failures of implementation. Therefore, additional research addresses application implementations that are similar to that of the UMMSE editor view. The ideas identified in related research are applied in this report to justify this editor view design. Figure 5.2 provides a pictorial of the UMMSE editor application view with the class scheduler UMM Specification (Appendix D) loaded.

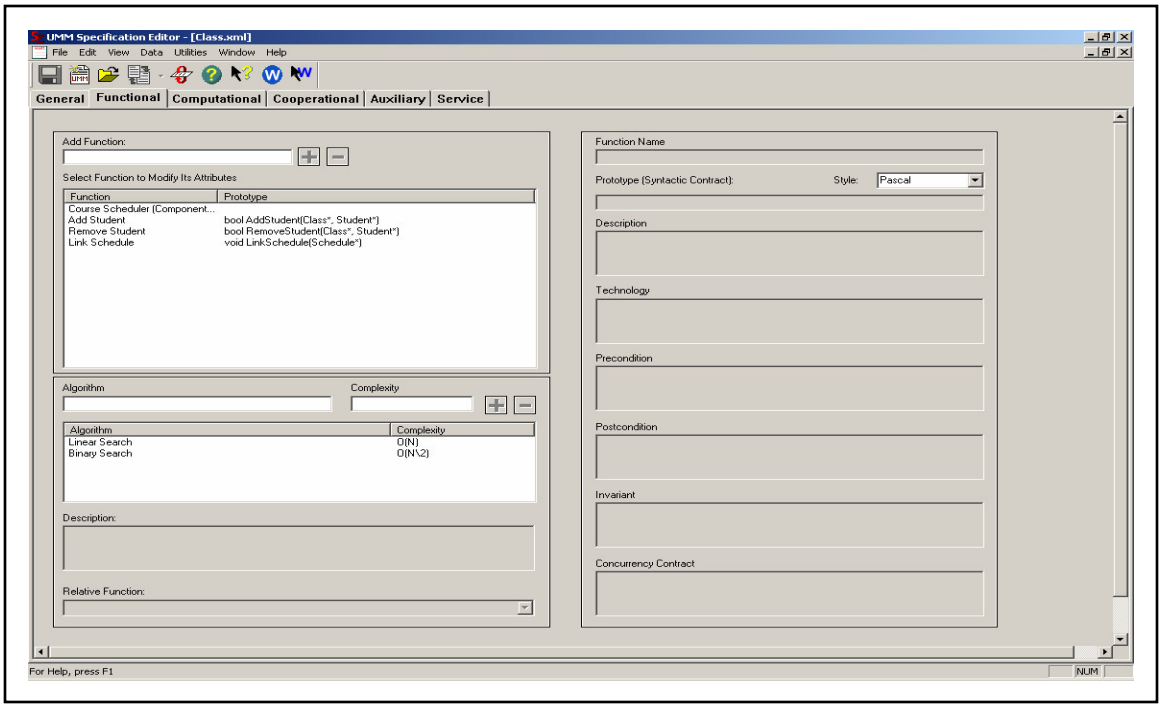


Figure 5.2. Editor Application View

5.3.1.1 Data Access

Data access and modification is the key functional requirement of the editor view. In particular, the data to be accessed is that contained by the formatted XML UMM specification. This function is accomplished using a dialog style user interaction interface. For the purpose of this section, a dialog interface is defined as an interface that provides user prompts using static textual content and variant user response controls, as opposed to definitions often provided in GUI design programming guides that refer to dialog interfaces as ‘pop-up’ interactions. Throughout the past development of dialog interactions multiple conceptual styles have been described and implemented. UMMSE uses a variant type of a dialog style referred to as a ‘form-fill’ dialog. A form-fill dialog provides editable or selectable points on the screen where the user provides the prompted information. According to Booth [BOO00], form-fill (free form-fill in the case that random keyboard is allowed) emphasizes the system control over other styles that may lean towards providing the user the benefit of control. With minimal input alternatives,

the user is granted little control; however the advantage of this approach is that the users' application knowledge and memory requirement are limited [BOO00, 50-51]. Additionally, the added system control allows the developer to limit the possibility of user error, particularly content error, which in the case of a strictly formatted document such as the UMM specification, is extremely important.

The dialog portion of the editor view combines the free form-fill style with a loose interpretation of direct manipulation style characteristics. Booth defines this style in that "user's actions should directly affect what happens on the screen", furthermore, indicating that this can be implemented to the extent that the user gains the feel of actually manipulating physical objects on the screen [BOO00, 51]. UMMSE implements this style in the form of a tab control. Each document (dialog) is displayed with multiple content tabs individually labeled, harnessing the interface design technique of metaphor mapping creating the appearance of layered pages on the screen, with each page consisting of particular data content. As each tab is selected the dialog page reforms to include the data belonging in the selected content sector, removing previous content. This change of display provides a sense of direct manipulation to the user, albeit a loose implementation of the style. The implementation of a combinational direct manipulation and free-form dialog style, on top of MDI application architecture, is uniquely challenging in the field of interaction design; and is a key point of this research and evaluation concerning its actual implementation of the UMMSE editor view.

5.3.1.2 Control Access

The control access is the second main interaction format required for the editor view of UMMSE. It enables the user to access all necessary UMMSE controls from a single access point of the editor view. This is accomplished using a menu and toolbar format located on the application interface, which encloses the dialog interface and completes the two-part editor view interface. There are three categories of control accessed by this interface. The first supports the MDI architecture and provides document control. Standard document requirements, open, close, and save, are available to the user

by a menu selection, toolbar icon, or both. The second category provides an extension to the data control interface by supplying access to edit commands such as copy, cut, and paste. These functions are used to modify internal document data, as well as inter-process data manipulation by use of an operating system interface (Clipboard on Windows™). The final control access category made available by the interface provides complete access to the functional components of the entire UMMSE application. This list is inclusive of basic functions, such as close and launch capabilities of UMMSE components, UMMSW, specification converter, and application help.

Using a combinational functional frame system consisting of an application menu and an iconic toolbar, control access is achieved. This system uses a ‘count to less than count’ relationship to maximize access while minimizing required user actions. This means the menu system includes a certain count of function and component accesses and the toolbar system provides equivalent access, except only for a number of accesses less than count. In the ideal situation, the ability of single action access as provided by the toolbar would be used for all selectors. However, this implementation would lead to an abundance of icons that would undoubtedly use too much screen real estate as well as clutter the display. Therefore, the toolbar is only used for those controls that are expected to be common. A menu and toolbar command access system enables user flexibility, though it still maintains strict restrictions on command combinations as well as command scheduling, preventing the user from executing commands that are invalid for the current system state [BOO00, 49].

The menu system requires two or more user actions to gain control (initial menu selector followed by at least one sub-menu selection). Each toolbar access control is also included in the file menu system for the purpose of completeness and consistency, providing the ‘count to less than count’ relationship. Since the menu system is implemented via a single bar combined with multiple pop-up sub-menus, screen real estate is only of temporary concern during the life of the pop-up. Furthermore, the menu system uses a particular access location system to adhere to the theory of maximizing benefit by minimizing user action. This is accomplished using a classification system that uses two two-pair classifiers. The first pair, ‘frequent’ and ‘occasional’, refers to the

expected access count per application use of the control. The second pair, ‘many’ and ‘few’, refers to the number of users who are expected to require access to the control. Isaacs suggests the following classifications for determining control access menu locale. Those tasks classified as “frequent by many” should be located on the main menu. Additionally, tasks that are accessed by many (frequently and occasionally) should be placed on the main menu, or initial sub-menu, as the screen real estate permits [IAA00, 135]. Other classifications should either be located on the initial sub-menu or subsequent menus with “occasionally by few” taking least precedence. Alternatively, Isaacs indicates that those controls not located on immediate menus may also benefit by mapped shortcuts to facilitate the access to frequent users of the control.

5.3.1.3 Context Help Design

The menu and toolbar systems provide methods of accessing the help content in a generic manner. Due to the technical content of the UMM specification and the unique interaction interface of the UMMSE, it is necessary to offer multiple access points to the help content. The feature specification of UMMSE terms the access and help content as contextual help because the help content can be accessed in direct relation to the current task. The editor view interface provides the interaction to accomplish this access. There are two direct methods that are implemented to allow the user access to specific help topics without any form of search or indexing mechanism. Each mechanism requires a single user action to access the intended help topic.

The first action requires the user to perform an action directly on the control to indicate that help content is required. There are three ways of implementing the interaction. The explicit approach requires the placement of a help icon (for example a question mark icon) in the near vicinity of the related control. The obvious advantage is that the action required by the user is visually explicit; however, this is a tradeoff with the intensive use of screen real estate.

The second approach makes productive use of individual tool tips. When a user briefly hovers the mouse position above a control a small tip with limited text appears on

the screen. Additionally, if the user requires further information, the tool tip may be clicked to access the complete help topic within the help system. In a study of the implementation of tool tips associated with the Explorer Bar in the Windows Explorer application, it was determined that this approach provided a positive tradeoff because of its implicit nature and minimal text display in favor of not requiring any additional screen real estate [BER00, 157].

The third approach is a hybrid of implicit and explicit implementation and requires two user actions to execute. First, the user is required to enable the contextual help feature; accomplished by depressing a question mark icon (or similar) located on the border of the editor view. Second, the user double-clicks on the control related to the help content that is required. This final action launches the help system opened in view of the particular help topic that relates to the control. All three techniques have their advantages and disadvantages UMMSE uses the third approach in its design, because it offers a compromise between the first and second approaches; as well as a complementing medium for the three user profiles.

5.3.1.4 MDI Relationship

UMMSE is a MDI-based user interface application and therefore supports multiple documents and visual representation of those documents. On most operating systems, the organization of multiple files and their external representation (frame representation) is highly regulated by convention. The initial implementation of UMMSE associated with this research project is developed based on the Windows™ operating system. Therefore, a brief explanation of MDI interface implications is discussed as it relates to this particular operating system.

First and foremost the document frame itself has multiple styles, minimized, user-defined, and maximized. A support for a minimized document requires that the document not be in dialog view, however, a representation of that file must be visible to the user in order to regain visualization. This can be accomplished either by listing the file name in a header-only frame within the application desktop or by an iconic representation of a

document. Both approach provides any true advantage or disadvantage to the user, and therefore, this decision is left to the implementing developer. Support of the maximized state is simplistic in nature. The document-containing frame should use all available application desktop for visualization. This provides the docked appearance of it being the only opened document (though this is not necessarily true).

The user defines the third style. In practice, this is accomplished by providing a resize control for the frame and is the most complex of the three styles. Resizing guidelines must be established by the application to define the degree to which the frame may be resized. Obviously, the document can be resized up to only the maximum available application space, which is equivalent to the maximized state. The true difficulty of this style is establishing the appropriate resize down limitations so that the document remains usable but frees as much screen real estate as possible. Because UMMSE uses a dialog style representation of the document, this style is complicated by the fact that each dialog is variant in its content. Therefore, each control of each dialog must have defined resizing actions and limitations in order not to distort the user's view and to maintain usability. UMMSE implements this feature by setting fixed sizes per control that prevents the user from resizing to a state of un-usability.

Two additional features are implemented by the UMMSE application to maintain usability throughout user actions of resizing and document toggling. Vertical and horizontal scroll bars are provided on each dialog frame to allow the user to bring controls into view that have been hidden due to window size constraints. However, a usability requirement exists that defines a horizontal scroll bar to only be valid in the case that the frame is in a less than maximized state. The exception to this stipulation handles the extreme case where the screen resolution setting is not capable of displaying the entire display in an appropriate manner. The second feature is implemented to aid the user in understanding current document location when toggling between multiple document views. The main application header bar maintains a currently opened and focused document representation by displaying the file name and view type. When a user brings a new document into focus (by executing toggling, new, or open commands) the display is changed to represent the new document that has been brought into focus. By

combining multiple techniques, each subtle in their own right within the interface design, usability of the editor view can be maintained while offering document view flexibility to the user.

5.3.1.5 Related Development

As XML has gained popularity over the past few years as demand for XML file parsers and generators grew. In typical fashion, as the demand grew, the software industry followed through and developed a multitude of products to meet the demand. In this analysis, three different applications considered to be (or to have a component) XML editors are examined from the viewpoint of usability and screen representation. Following the introduction and analysis of the three applications, the derived information will be applied the UMMSE application. It is important to reiterate an important UMMSE usability specification before continuing the analysis. A unique approach, which is intentionally not represented in the sample set of studied applications, is that the specification editor is a complete abstraction from the XML storage format. Users of the UMMSE application are not expected to have any knowledge of XML. Similarly, the UMMSE application does not intend to expose the user to any particulars of XML and its implementation. The UMMSE application, and UniFrame system, uses the DTD file only to define the format and content placement of the XML implementation. Its integration is hidden from users of the system and does not have a direct impact on a user's view of the editor interface.

The first application analyzed emphasizes capabilities to edit the XML document as well as its schema (DTD), Pierlou Visual XML [PIE00]. The application is geared towards XML educated users, as it makes no attempt to abstract the details of the XML file from the user. It offers two particular dialog style interfaces, one to edit the XML file and the other to modify the schema file. Pierlou Visual XML uses a standard tree style structure to display the entities of the XML file. From this interface a user has the ability to change the definition of the DTD file, which in turn defines the valid structure of the XML element data of the XML document. UMMSE requirement specifications do not

require explicit access to the DTD. Therefore, this particular implementation choice is not appropriate for use within the UMMSE application. However, the implemented tree control style display does offer a possible adaptation to UMMSE's secondary editor view that displays the actual XML document. This display will not offer editing capabilities in order to limit the user from erroneous input. However, it is feasible to offer a similar view as an extension to the context help system of UMMSE. This implementation would allow the user to select a data element from the tree control to invoke a system response of displaying logical help concerning the selection in an adjacent text window. Using this adaptation, the user is provided an explicit view of the XML document and the capability of directly referencing element related help, while maintaining the required XML abstraction by not allowing the document to be edited directly from the displayed interface.

IBM's Alphaworks Xeena application [IBM00] offers a divided representation of the XML document attributes and their subsequent values. This implementation of the editing feature more closely resembles the abstraction goal of the UMMSE application. However, since the application is still generically associated to general XML documents, it doesn't provide a complete abstraction. The free form edit capability of editing an element's value directly is similar to the free form text fields offered on the UMMSE editor interface. For example, within this application the *Name* attribute could be edited simply by altering its value in the relationship window. Similarly, the UMMSE application implements this capability as well; only that it explicitly offers the edit box labeled as "Component Name". The UMMSE takes this action one step further towards complete abstraction can be seen using a non-free form example. The *Mobile* attribute is a boolean value. Using the Xeena editor, the user would have to add either 'true' or 'false' to the value edit box. The possibility of user error exists because other free-form values could also be added, for example 'unknown' (ignored in this analysis is that, in practice, this application is capable of justifying this data against the DTD to avoid this uncertainty). Because the UMMSE application is geared towards a specific XML document, the UMM specification, the UMMSE interface provides a complete abstraction by limiting the interface to a toggle control with possible values of 'true' or

‘false’. This particular relationship between a free-form edit control and its attribute does offer a secondary implementation that may be valid for short-term extensibility of the UMMSE application. Specifically, it may be beneficial for UMMSE to offer an explicit new attribute and value relationship box to allow users to add elements that are not in the current XML schema version known to the application. It is appropriate to revisit such an idea after the operation integration phase to determine its feasibility.

The third application analyzed in this section is a web browser that incorporates a XML viewer as one of its many valid document types. Microsoft’s Internet Explorer™ uses a direct translation approach in representing a XML document providing the least abstraction of the three software applications. Its advantage is that it provides complete visual representation of the XML document to the knowledgeable user. Internet Explorer™ (6.0 and later) uses a collapsible tree control for the main user interaction point. From this control the user is able to expand or collapse any particular hierarchy of the XML document. This is a natural interface design choice for displaying a XML file exploiting its hierarchical characteristics.

Readability is a key concern for the secondary XML view provided by UMMSE. It provides a mechanism for the user to preview the final data file output before it is actually written to the disk. Using a hierarchical tree control provides a distinct and important usability goal to the XML view interface. A XML document can become quite large, and most often will require more than a single screen space for full display (this is expected to be the case with a UMM specification document as well). Thusly providing a collapsible interface where the user is able to identify the desired sub-category for viewing, as well as maximizing use of screen real-estate while externalizing clutter data (data currently not being viewed) in a hidden fashion. Disadvantages exist in such an interface and should be understood before implementation. In understanding possible user issues, a designer may be able to maintain the advantage of the interface while overcoming shortcomings. A tree control style XML interface is geared towards knowledgeable and expert users. Foremost, the interface inherently reflects the XML file and may be confusing to an unknowledgeable user. Experience and help documentation are possible solutions to this issue, however the interface itself may also be used to

educate a user. The interface implementation can be designed to infer that the different hierarchies are collapsible. Additionally, appropriate use of indentation can provide further inklings towards understanding the XML format.

Analysis of past development and commercialized products indicates that the primary use of XML notation to this point is for inter-process communication. Due to this, it is understood that the majority of users editing such documents are familiar with XML notation (and software development in general). The UniFrame project (facilitated by UMMSE) uniquely extends XML to be used in combination with human users. The three analyzed applications, as well as most available applications, concentrate on direct editing of DTD and XML files. Therefore, there is little previous research and implementation to adapt or discourage in the design of the UMMSE editor interface. Research and development of the UMMSE project provides one approach and implementation of XML abstraction within cooperative user interaction. Referenced applications offer a variety of methods for modifying, creating, and reading XML documents. Without concrete user analysis in relation to the applications, success of their implementations cannot be determined, though conceptual advantages, disadvantages, and interface features are analyzed in this research in order to provide additional design considerations for UMMSE.

5.3.1.6 Editor Justification

Application and research history of the use of XML documents indicates that the necessity to abstract the XML structure from the user remains uncommon. The typical use of a XML document in the industry is for inter-process communication. Applications on the same machine, or on different machines, use the structured format of XML to share specific information. The UniFrame system also uses the XML formatted UMM specification in this manner in the form of transferring data from a component owner to a repository, and then on to a headhunter server. However, the system also requires its use in interfacing with its human users. The UMMSE application facilitates this interaction by providing an intermediary interface to indirectly create and modify UMM

specifications. The flexibility and cross-platform exposure that the XML provides ensures an expanding realm of uses for technology. In certainty, use of the format to abstractly transfer information from human user to application will grow.

UMMSE justifies its use of a combination dialog and MDI architectures by referencing its developed usability specifications. Specifically, the editor interface must completely abstract the details of the XML document from the user. The exception is the secondary viewing interface that provides a non-editable representation of the true XML data structure in hierarchical view. Disadvantage analysis of typical XML document editors provides further justification that the goal of abstraction is significant within the UniFrame environment. Subtle adaptations of proven features, such as tree control views and free-form attribute editing, ideally fit the UMMSE model of goals and are therefore beneficial in implementation. This section's analysis of the interface styles MDI, free form, menu systems, and direct manipulation, concludes that in combination the styles can provide a unique user interface application. This approach is used as basis for the development of UMMSE's editor view interface.

5.3.2 Wizard Application View

"Provide multiple access to information: Access to information should not be restricted to one particular method; instead, many possible ways should be offered to get a specific piece of information" [SCH01, 468]. This point is made in justification for developing a new and more advanced method to browse illustrated documents. In an editor environment, this idea can be applied to an application's offering of methods available to edit pieces of information. Precisely what the UMMSW application provides to its users is a second formative mechanism for creating and editing specification documents. UMMSW is a derivation of the main editor view and editing functionalities. The derivation's functionalities are consistent with the editor view. However, its user interaction style is drastically different. The extension uses a now common style of interaction popularized by Windows™ applications, and in particular, applications developed by Microsoft for their operating systems. This section emphasizes the benefits

of the Wizard extension and provides an analysis of its requirements. In addition, a brief history of the wizard style compliments analysis of popular application's that have used various wizards.

5.3.2.1 Data Access

The wizard view is an interactive extension of the UMMSE editor view. Functionality and data access as it pertains to an individual specification document that the editor provides does not change. However, the wizard offers it in a more compact and directive manner. Contrary to the editor interface, the wizard interface is a single document interface (SDI). This is because, at any given time, only one specification document may be opened for editing within the view. Due to this, the interface only has one possible set of related data. However, like the editor view, there may be multiple data access points per interface representation.

5.3.2.2 Interaction Environment

A main goal of the wizard is to provide a secondary method of editing specification information. However, for the new method to be truly beneficial, it must offer extensive differences between itself and the editor so as to provide additional interaction sequences to the user. The UMMSW user interface is designed for two particular interaction user groups. The expected use ratio by user groups will be discussed in detail; however, it is first necessary to discuss the interaction model of the wizard.

Analogous to a graphical storyboard, the emphasis of a wizard is to provide a sequential functional mechanism. In the case of the UMMSW implementation, the functionality mimics that of the editor and provides a method of specification data entry. The intent of the wizard expands upon two aspects the editor view exhibits, though not in a prominent fashion. First, a wizard adds a logical sequence to user actions. The editor view infers data sequence by use of the tab style, however once a user selects a particular tab only visual sequence (screen layout) is provided and very little control is maintained

by the application. This is acceptable in the context of the editor view because its prominent user type is that of an intermediate or expert status. However, since the wizard is geared towards a basic user, it is necessary to provide sequential steps (pages) so as not to overwhelm a basic user of minimal knowledge with the UMM specification and its data relationships. Figure 5.3 provides a representation of the first three pages of the UMMSW.

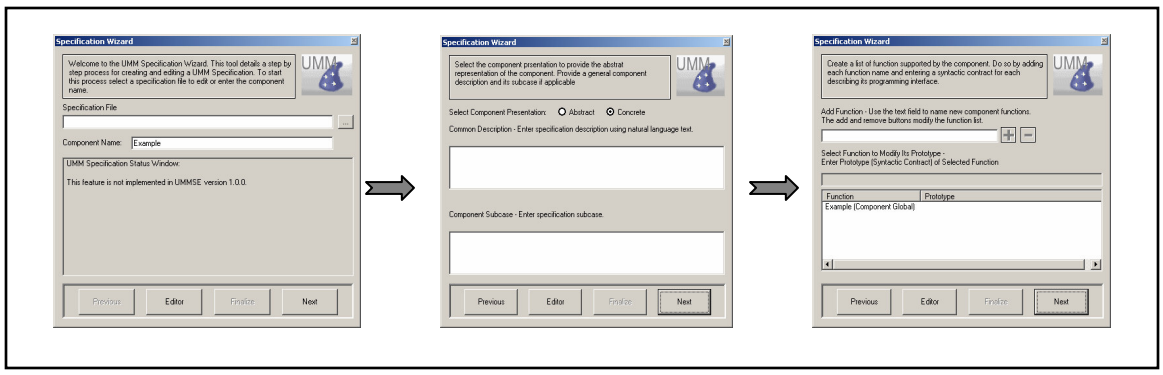


Figure 5.3. Sequential Wizard Views

For consistency, the user interface design of the wizard maintains a similar representation throughout all sequential presentations. The design incorporates three major screen spaces that make use of logical display to further ease the user's action. At the top of the dialog window, an information header space is maintained throughout all visualizations. This header provides a common place for the application display directive information. The header is refreshed on each page to describe the data content of the page, and to provide any further information to guide the user. After reading the directives, the user's eyes enter the data entry screen space. This space is refreshed on every page to provide the next logical block of data required. The visual controls of this particular part of the interface are consistent with the editor view. The difference is that a block of data that is represented by six controls in the editor view may be represented by three controls on two different wizard pages. This tactic decreases the presentation of data to the user and allows the application to maintain greater control over the user's input. At completion of data entry, the user's eye movement naturally progresses towards the lower half of the dialog.

The final screen space is located at the bottom of each page and consists of four functions: previous, next, action and editor. A navigation-style iconic button that maintains an availability state represents each function. The state is set according to the user's progress through the wizard and local progress through the current page data. The previous and next buttons provide the user with wizard navigation ability; they change the page respectively to either the previous or next content page. There are two factors in determining their button statuses. Obviously, if a page is either the first or last of the sequence group then their respective previous and next functions will not be available.

The second factor is determined by the status of the local data content and entries. If a required data control field has not yet been edited then the next button will not be available. This indicates to the user that further input is needed, and provides a controlling mechanism to the application designer.

The third function, applied by the editor button, provides the user an ability to return to the editor view (closing the wizard). Generally, this function is always available; however, there may be rare cases where the user must input functional control data before returning to the editor. In this specific case, the button maintains a state of inactivity to indicate its status to the user.

The fourth button is a generic action button. The functionality enabled by this button changes depending on the specification edit state and the data entry states. For example, this button may represent a 'complete' status, in which case the specification is in a state where it can be finalized. If the user were to invoke this action, the application would prompt for completion information and proceed to execute the specification generator module.

5.3.2.3 Related Development

The wizard design can use many different conventional graphical interface representations. Additionally, it is also possible to combine multiple techniques to create an interface that conveys an application's intent in the best possible manner. In evaluation of multiple wizard application extensions, two were chosen for this in depth analysis.

Each of the chosen applications uses a different interaction mechanism to accomplish the task. One application, Network Setup Wizard (Windows XP™), uses the sequential style described in the previous discussion. This is used as a comparison with the UMMSE implementation because the employed interaction techniques are similar. The second, Microsoft Visual C++™, uses a different interaction technique and is used as a comparison of an opposing implementation.

Introduction to the opposing technique is provided first in order to develop an understanding of how wizard tasks may be completed using different interfaces. Microsoft Visual C++ uses a wizard dialog to provide the user with a mechanism to create, expand upon, and link class objects (and their attributes) of a particular programming project. The Microsoft Visual C++ application refers to this extension tool as the *Class Wizard*. Microsoft developed the interface using a tab style interface. Each tab represents a different relationship of the classes and their attributes. The application provides this tool to the user as a shortcut mechanism for coding common relationships within a C++ program. For example, a user may launch the wizard to create a new class. To accomplish this task, the user selects a *new* function and the wizard responds by providing a secondary dialog requesting additional input related to the task. At completion of entering the data, the user accepts the changes. In response, the application creates the new class within the program, generates the necessary basic relationships, and generates the code template files and adds them to the current project.

Though the tool interface is explanatory, it is far from self-explanatory. A new or basic user of the application must reference the help contents in order to initially learn the task at hand. As stated previously, the intent of UMMSW implementation is to provide a simplistic tool and learning environment. For this purpose, the sequential implementation is a better fit and one of the deciding factors not to emulate the Visual C++ implementation. By using a tab control within its interface, the *Class Wizard* is able to offer complex tasks to the user. However, this also adds additional complexity that does not lend itself to a learning environment. A tab control requires that a secondary interface be used to extend upon any particular function. As mentioned in the example, once a function is selected, a secondary window is provided to continue with the task. This

complicates the visual screen environment by layering multiple windows and does not provide a navigational property. A user must inherently navigate multiple windows to accomplish a task, requiring the user to become familiar with multiple interfaces rather than being presented a comforting quality that using similar or the same window can provide. This is another reason why a similar interface structure was not used in the development of the UMMSE implementation. Finally, the tab control style too closely resembles the tab interface architecture used in the UMMSE main view. Mimicking a similar environment within the application extension would not provide the 'multiple access' goal of representation for which the wizard is designed within the UMMSE application. Implementation of this style would not provide enough additional user benefits to justify its inclusion.

Since the UMMSE Wizard is designed to accomplish one task, that is specification editing, the standard wizard sequence implementation is a natural fit. Windows XP™ uses a wizard to provide a consistent platform for many of its administrative configuration tasks. *Network Setup Wizard* is an example interface that is consistent with other wizards on the XP operating system. It makes use of the sequential interaction design similar to that discussed and implemented within the UMMSE application. The previous and next state buttons direct user progression through the task towards completion. A common header is maintained throughout the multiple interface representations, each header clearly describing the required page input. Within the page contents, additional information fields are provided as necessary to properly direct the user in completing the page's fields. The interface design is conducive to a user-friendly environment limiting the possibility of user error. The wizard's simplistic (narrowed) data control fields also aid in limiting task exposure to the user. This allows the user to think in a linear fashion and discourages the possibility of overwhelming the user. Unlike the *Class Wizard*, the Windows XP™ task wizard is simplistic and intentionally directed towards beginner users. At the same time, its interaction design provides an efficient task completion interface for more advanced users.

5.3.2.4 Wizard Justification

Many variations of wizard applications exist beyond the two discussed previously. *Class Wizard* and *Network Setup Wizard* were selected because they present opposing complexities and approaches towards task completion interfaces. Analysis of the applications presents multiple Windows' conventions that will be integrated in the UMMSE integration as stated by the user requirements. The UMMSE wizard extension is based on interaction design concepts used by the *Network Setup Wizard*. Based on the provided analysis, its sequential and simplistic approach seems most appropriate for the interface design goals of the wizard integration.

One functional design decision does deviate the UMMSW integration from the standard Windows XP™ wizard. The UMMSW interaction design environment includes two additional functional buttons. This extension adds minor interaction complexity. However, it is a necessary compromise to avoid extreme complication while maintaining the flexibility required to edit a large document through a wizard interface. Presenting the large editor view within the wizard style requires the division of five standard editor pages into twenty wizard pages.

Therefore, it is necessary to add a third navigation dimension not included in standard wizards, which provide only previous and next movements. This dimension allows a user to leave the wizard (switching the document to an editor view) or enter the wizard at virtually any wizard page. When the user presses the button labeled 'editor', the view is returned to the main editor. Depending on the specification's status, the user may be queried for input to determine if the edited data should be saved and transposed into the editor view. Furthermore, the UMMSW adaptation contains an entry point on each page. This means that the user may choose to transform a document in editor view into the wizard view, and do so by starting at a particular data entry point that is not necessarily the starting point. Acting as a guide, this option may be chosen by a user that has become confused, or as a secondary level of help before accessing the context help database. The second additional navigation button is designed to be a catchall action button. In the common case it represents an imitative editor button; however its

functionality is a bit more specific as it launches the finalization execution (save & continue) before returning to the editor view. If necessary, this button may be used to represent any other action that is deemed appropriate within the interface. In combination, the analyzed beneficial attributes and extended UMMSE integration options provide the user with a practical and simplistic editing interface.

5.4 Data Interface

The application's two main user interfaces, editor view and wizard view, require that a structured set of data be transformed into an operating specific encapsulation of display and input controls. This will be accomplished by using a layered software development model as specified in this section. A layered approach brings multiple advantages to the development project. The concept of using layers introduces object orientation that in theory allows each object (layer) to be developed independently. Additionally, the layered model allows development to be completed in a non-linear manner. This allows interface development to be continued without the completion of supporting, though independent, layers. Layers also provide a conceptual abstraction of the data transformation process, which is important for code maintenance and new feature support.

Four specific layers are used to represent data transformation support for UMMSE's user interfaces. Figure 5.4 represents this transformation process, which is bi-directional. Each layer requires a form of data as its input and produces a form of altered data for its output, resulting in five data forms. In linear combination, execution of each of the four layers transforms a raw data format (XML specification) to an operating specific interface definition. The interface definition is composed of input and output controls that provide communication paths between the user and machine interaction. Because the final layer's output provides the visual display, it must be operating specific. However, this layer should be as thin as possible since it must be converted in its entirety when porting (Section 6.4) the UMMSE application between different operating system

platforms. Each layer and data form is defined in the following sections explaining the layered approach within the UMMSE environment.

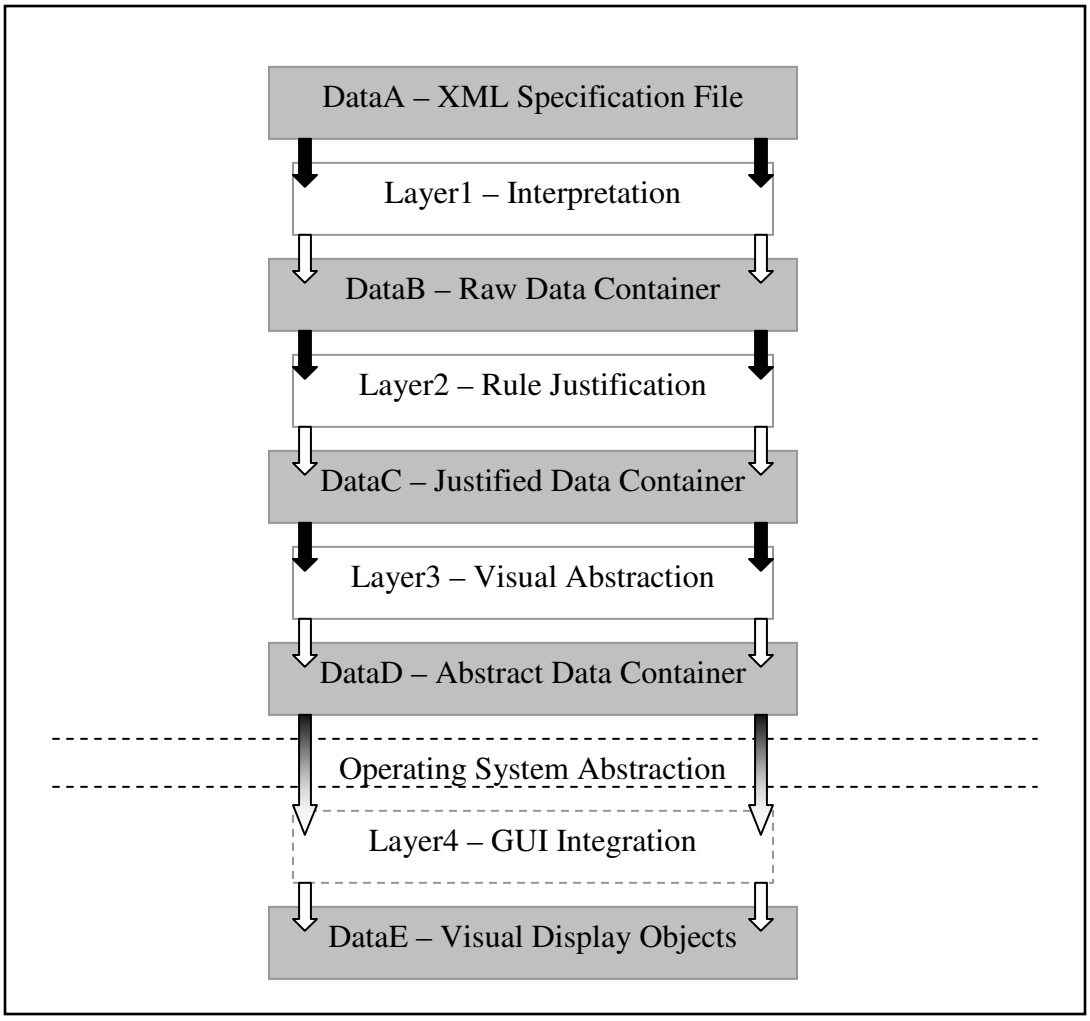


Figure 5.4. Data Object Design

5.4.1 Layers

Each data design layer represents the transition period between two different data object states. During the design phase, each layer is simply a logical description of actions occurring on a specific set of input data. There are four data layers that are identified in the UMMSE design: Interpretation, Rule Justification, Visual Abstraction,

and GUI Development Integration. Each layer provides a specific function in creating the user interface design detailed in this section. The final layer is operating system dependent. As the initial development of the UMMSE application will be on the Windows™ platform, this layer and its final output will be explored using the Windows™ operating and development environments as a basis.

5.4.1.1 Interpretation

For the purpose of this description, only the transformation of the interpretation process will be explored as depicted in Figure 5.4. It should be noted that the generation of data is simply the reverse of the interpretation process, and replaces the interpretation phase when outputting a UMM Specification. This layer takes as its input a raw XML file that is formatted to the UMM Specification definition. The XML is parsed (by the interpreter) and each field of data is stored as generic type with hierarchy location ID. Its output format consists of data groups of basic types that are known by the UMMSE application. The known association is generated by template compilation that takes into account all XML specification format templates (DTDs) known by the system. Appendix C provides an example DTD template that would be used in this compilation. The interpretation layer is considered to be raw, meaning that it's input and output is in a raw format. Logical error and interpretation error checking is not performed at this layer. If interpretation cannot relate a piece of data to a known container from compilation it is simply ignored.

5.4.1.2 Rule Justification

This transformation layer provides error correction and data simplification to the layered approach. It accomplishes this by applying rules defined by templates maintained by the UMMSE development group. The rules are applied to the raw data container objects to create the output containers for data that have been rule justified. The rules can result in two different actions being taken on the data. First, data that does not fit the

profile of its owning tag as defined by the rules will be removed, returning that data element to a status of null input. The second function of the rule justification is to transform raw data into logically correct data. The logistics are provided by the defined rules. The data format is inherently hierarchical by the implementing XML format. Due to this, it is possible for sub-branches and values to have different meaning and validity depending on the value of an element of higher hierarchy. For this reason, the rule justification must be able to determine the validity of correlated data. If data is determined to be incongruent with the rule definitions, the offending data is returned to a null input status. As a result of these two functions, the resulting justified data container is correct in format and content as specified by the governing organizations.

5.4.1.3 Visual Abstraction

This layer creates a link between data and its visual representation. Additionally, it defines each individual data element's interaction attributes. At the completion of this layer, a system is capable of recognizing an interface's complete content and conceptualizing its layout. This layer makes use of two input parameters. First, it accepts rule-justified raw data containers from the rule justification layer. Secondly, a compiled template of interaction attributes is referenced to create each attribute's interaction control type abstraction. Just as at the rule justification layer, this template is maintained by the UMMSE development group. It is feasible that the same template could source functions of each of these two layers. Template data is combined with the provided raw data to form abstract display containers to be implemented in the concrete display interaction.

5.4.1.4 GUI Development Integration

This layer implements the operating specific integration of abstract display containers into concrete development environment controls. Each display container control type is built into an organized interface layout following specification as outlined

during the feature specification phase. At completion of this phase an operating specific description of the UMMSE editor view will be established and capable of being executed into the display of the UMMSE application (referred to as Visual Display Objects). From this point the user will be able interact with the XML specification's data and execute the application goal of being able to view and edit a valid UMM specification. When editing is complete, this process is reversed instantiating the generation form of the bi-directional transformation.

5.4.2 Structures

Data structures provide the input and represent the output of each conversion layer. Each has a concrete implementation equivalent, which is discussed during the integration phase. The design of each data entity describes the logistics and purpose of each of the five data structures: Specification File, Raw Data Container, Justified Data Container, Abstract Data Container, and Visual Display Objects.

5.4.2.1 Specification File

Represents a UMM component (defined as in a specification) interface in the UniFrame system. This file is created by component developers, stored in repositories, and accessed by headhunter services. Its format is known and defined by rigorous rules and is stored in the XML file format. UMMSE interprets and generates this file as its service to the UniFrame system. Information presented in this file is directly reflected in UMMSE user interfaces and facilitated by the layered approach.

5.4.2.2 Raw Data Container

The raw data container is a cached version of the XML specification file. It is a representation of all valid tagged data that is stored in the process' memory space. It is composed of data placeholders of string type. The organization of the data is generated

from compiled templates that dynamically define the structure of the data. In a simplified implementation, the raw data containers can be static and have a one-to-one correlation to the latest version of the standard specification template. In a static implementation the application's knowledge of the DTD template is known before run-time execution. Of course, this implementation adds to the burden of maintaining the interpretation layer and raw data container structures when the DTD is modified.

5.4.2.3 Justified Data Container

The rule-justified raw data container is a complete internally structured representation of all valid data contained in the original XML specification file. It can be viewed as a correct revision of a raw data container with complete data accessibility. Justified containers also perform the structuring of all composite high-level (UMMSE level) data structures. For example, if a particular representation of data uses a paired format, the rule-justified raw data container will compose the structure from the basic type data. As it is an extension of the raw data container, it also has the possibility of being implemented using either dynamic or static structuring.

5.4.2.4 Abstract Data Container

Abstract data containers are interaction abstractions for raw data containers. Essentially, they consist of the justified raw data plus additional attributes and data added by the visual abstraction layer. The main attribute applied is the specification of control abstraction type. This type is determined by predefined mappings between data types and a defined set of input and output control types. Because the abstract data containers are above the operating system abstraction layer they are defined in non-operating specific generic terms. The control type is complemented with a set of defined display attributes specific to the control type. Abstract data containers represent the last transformation of the data before they are applied to operating specific GUI development integration.

5.4.2.5 Visual Display Object

Visual display objects are coded and compiled object-oriented GUI controls. They are compiled into the editor view capable of being executed within the UMMSE application. All data required to complete this interaction is included within the encapsulation of the object and may be executed.

5.4.3 Data Example

This section presents a concrete example of the implementation of the four-layer data abstraction approach as used by UMMSE. The example follows the Mobile attribute from its initial state in the UMM specification follow through to its final state embedded in a visual display object.

1) XML Specification File:

<AuxiliaryAttributes Mobile=”true”></AuxiliaryAttributes>

2) Layer1 – Interpretation

Mobile value “true” is read from the specification file and stored in its raw data form: string value “true”, attribute value “Mobile”, and parent value “AuxiliaryAttributes”.

3) Raw Data Container:

{AuxiliaryAttributes, Mobile, true}

4) Layer2 – Rule Justification

Processing checks are performed on the AuxiliaryAttributes and Mobile tags to verify that their relationship is in agreement with the specification version. Also, the value of ‘true’ is verified to be valid for the attribute.

5) Justified Data Container:

{AuxiliaryAttributes, Mobile, true}

6) Layer3 – Visual Abstraction

Maps the attribute Mobile to a Boolean toggle control. Process also references a template map to determine the control's initial state values.

7) Abstract Data Container:

Control Type – Boolean
Value – true
Context Help ID – 102
Display Text – "Mobility Status"

8) Layer4 – GUI Development Integration

At this point, the operating system abstraction is dropped and controls transform to implementation specifics. In this case, a toggle control maps to a Visual C++ control CButton (radio button style).

9) Visual Display Object:

Psudeo Code:

```
CButton* pMobility =
        CButton{
                BOOL m_bValue = true;
                CString m_strDisplayText = "Mobility Status"
                DWORD m_dwContextId = 102;
        }
```

5.5 Overview of UMMSE Design

This chapter discussed the design used in the UMMSE application. Integration factors presented by the UniFrame environment and the applications were first analyzed to determine their impact on the design. In making this determination, a MDI architecture is selected as the main application structure. This chapter continues in designing each of the applications views. The designs emphasize their data access points and user control mechanisms, referencing previous works that factored into the designs. This chapter concludes with the internal data model design. This four-layer approach to design is a bi-directional transformation process that transits data between visual display and data file end points.

6. IMPLEMENTATION

This chapter focuses on the implementation of the project's requirements, specifications, and design to create UMMSE. Section 6.1 describes the coding environment used to develop the concrete application. Section 6.2 discusses the implementation of the two main UMMSE GUIs, specifically the editor and wizard views. Section 6.3 discusses the implementation of the layered approach used to model the UMMSE specification data. Code abstraction layers, and how they are used within the application, are discussed in section 6.4 to conclude chapter 6.

6.1 Source Code Environment

The application version of UMMSE designed and implemented for this project is based on the family of Windows™ operating systems. C++ is the high-level language used as the implementation language for the entire application. Though the source code may be built with any valid C++ compiler and linker, this project uses the Microsoft Visual C++ (VC++) environment. Included with the deliverables of this project are the source code and project space file required to load the UMMSE application code into this environment (compatible with VC++ 5.0 and VC++ 6.0).

Besides the standard C and C++ libraries, the source code includes two additional libraries (which have become standards themselves). The Microsoft Foundation Class library (MFC) is used in developing operating system specific code, which accounts for the majority of the graphical user interfaces and underlying document architecture. The second library used is the Standard Template Libraries (STL). This collection of libraries is used for its data container objects. Its primary use in this application is to provide complex data structures for code that resides outside of the operating specific code.

6.2 Graphical User Interfaces

As discussed in the specification and design phases, there are two main graphical user interfaces (GUIs). The application editor represents the main MDI viewing architecture and is therefore the focal point for the application. It is comprised of multiple integrated modules, which are discussed in the following section. UMMSW is comprised of one single user interface with multiple data views. It is only accessible from the editor module. However, its editing functionality is isolated from the rest of the application. This section continues to discuss class objects developed for this application and their methods, which underline the visual interface functionality. At the completion of the discussion a class organization tree is provided.

6.2.1 Module Relationships

Three major implementation modules comprise the main UMMSE view (editor): document window, toolbar, and application menu. The UMMSW consists of one major implementation module, the data view. The sections on the functional specifications and designs discussed the specific features that are supported by each module. However, it must still be determined how they are to support these features, a question addressed in this section.

6.2.1.1 Document Window

The document window represents the main viewing window of the application view interface. It includes a visual representation of the currently in view UMM specification data for editing. Two document input requirements must be satisfied for this window to be active. First, a UMM specification file must be opened in the application. Second, the UMM specification data must be divided into some form of hierarchical representation that determines the content of the window. In the first version of UMMSE

(applied in this project), this representation is static and adheres to version 1.0.0 of the UMM specification as defined in chapter 4.

The viewing window should not make use of a horizontal scroll bar and should only use a vertical scroll bar when absolutely necessary. Additionally, it is expected that the viewing window will be required to contain an abundance of information. Therefore, data input organization and grouping become key concepts that must be considered throughout the implementation process. Data that are similar in content should be organized into groups that visually imply that the data is related. An appropriate balance of white space, boxing, and graphical division should be used to convey the visual representation. If the importance of data can be ascertained, it should be conveyed within the window by displaying those of greater importance towards the top of the window.

The data display window also shares its relationship with the encapsulating data display window toggle controls. These controls enable a user to quickly toggle the data display window between multiple document views (editor and XML). Therefore, it is important that this window be visually comparable among the multiple views. To accomplish this, a grid mechanism (provided by VC++ environment) is used when implementing the interface display. Using a grid provides agreement between window borders, group controls, and display text of the multiple window views.

6.2.1.2 Toolbar

Provides a single input control to execute particular application functions. Each button on the toolbar represents a particular function using icon representations of the functions. The representative icon provides the second important relationship. It develops a relation of a function traditionally displayed as text to a graphical representation. A toolbar provides an outlet for immediate system feedback to the user. A function that is not currently available for execution can be visually disabled on the toolbar. When the user provides a form of input that now makes the related function available, the system can immediately activate the toolbar option, providing instant feedback to the user. Deactivating a currently active option can represent the opposing sequence of events.

6.2.1.3 Menu

Provides a single area of selection where all application commands can be accessed. The menu has implications on both the document view and the toolbar during the integration and implementation phases. It provides the user with file access controls that directly launch execution sequences on the document view (and the underlying document data module). The toolbar commands are also replicated within the menu environment, creating dual points of access. The implementation phase must propagate agreement between the two so that the user's perception remains that of one executing command.

The application menu also provides a feedback mechanism similar to that provided by the toolbar module. By limiting the user's selection control, the menu can effectively communicate the current state of the application and its data. Indirect representation, such as this, is a key implementation factor because of its vast integration impact. Correct selection states may come from and software module in the system. Therefore, consideration for application selection states must be provided in the development of each GUI class object.

6.2.2 Class Objects

Following is a list of major class objects that are used in the implementation of the UMMSE GUIs. Each listing briefly describes the functional and structural purposes of the class design. An organizational derivation chart (Figure 6.1) provides a visual representation of the class inheritance and interoperation. Furthermore, section 6.2.3 addresses key method operations that support application and functional control.

- CMainFrame : Implements the main visual frame of the application. This includes housing the application menu, toolbar, and status window. Its sole functional purpose is to handle default status messages. The default state of

the application is defined as a NULL document state; in this state no document (child) views are currently active.

- CTabMdiChildFrm : Implements the child form for the editor view frame. This child form controls the major visual interface, which is a tab control. Its sole functional purpose is to handle the user notification messages that are generated by the system when the tab control is toggled. Upon receiving such a message, the form executes a switching function on the editor form that proceeds to change the displayed visual controls.

- CDocViewFrm : Is a base class for the two major form views of the application, editor and XML views. Its sole functional purpose is to handle status and notification messages that are shared between the two views. Such a responsibility is to respond to the user's mouse 'hits' (hit control) when either the context help or context wizard execution modes are in effect.

- CEditViewFrm : Implements the editor form view. This view handles user notification messages generated from execution of the currently in view dialog controls. These controls represent the UMM specification editing capabilities provided by the application. It inherits from the CVisualControls object, which performs the majority of the executing editing commands.

- CXmlViewFrm : Implements the XML form view. This view provides a visual representation of the UMM specification in its native XML format (tree hierarchy). This is purely an informational user interface and no editing functionalities are permitted.

- BaseControl : Implements a generic interface between visual controls and their representative abstract data containers. Included generic interfaces are: UmmCtrlBool, UmmCtrlInt, UmmCtrlFloat, UmmCtrlString,

UmmCtrlListInt, and UmmCtrlListString. Each interface implements Get, Set, and Remove operations for their data. The implemented base control used is a direct reflection of the type of data used for a visual control. For example, a list of data values presented in a list box uses an UmmCtrlListString class.

- CVisualControls : Implements the execution methods for all user edit actions shared between the editor view and the wizard dialog. Its sole functional purpose is to accept notifications reflected from its parent (CEditViewFrm or CWizardDlg) and performs the desired actions. A notification is reflected when the acting object, the parent, does not have a function to process the notification. Actions include data container modification and GUI updates. This class houses all base controls for the views and implements their data modification methods to perform data container modification.

- CWizardDlg : Implements the UMM Specification Wizard dialog interface. It reflects shared user notification messages to its CvisualControls base class. Additionally, it handles user notification messages that are specific to just the wizard interface

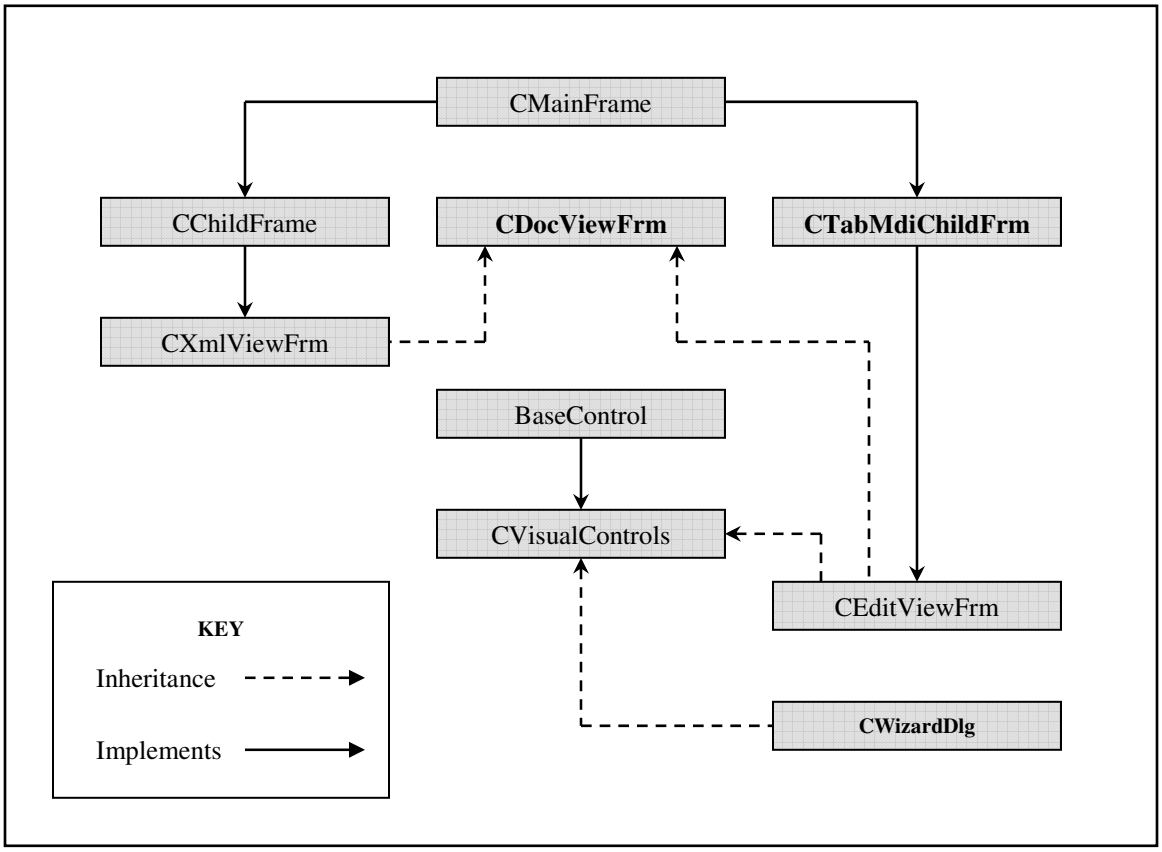


Figure 6.1. Graphical User Interface Organization

6.2.3 Method Operations

The previous section noted general functionalities provided by the major classes of the graphical user interfaces. This section further discusses the implementation details of these features. To do so, a listing of major class methods is provided. Each method is described by these characteristics: identification using C++ class scope syntax (::), description of functional purpose in natural language form, and applicable method implementation details.

- CDocViewFrm::OnHelpHitTest() – This method is executed when the user clicks the left mouse button while this application is in either context help or context wizard mode. Upon execution it makes a determination of the

application's state and identifies the context id for validity (provided from mouse position). If the proper conditions are met, then either the context help or context wizard supporting dialogs are launched.

- CEditViewFrm::SwitchTabView() – This method is called by an application class that wants to change the page in view on the main editor form (the tab control). Upon execution, it invokes the display manager, providing it a parameter page ID. The wizard manager adjusts the active controls using a defined mapping of page IDs to control Ids. Following execution of this method, the user's view of the editor form's control contents has changed.

- BaseControl::Init() – This method initializes the abstracted visual control. The initialization requires that an abstract data container, high-level hierarchy tag, and UMM specification tag be provided. These three pieces of information allow the control to uniquely identify the data for which it provides the visual interfacing.

- BaseControl::UpdateData() – This method controls the member data of the base control. Its core task is determined by a boolean variable passed by the calling entity. Upon execution, it either invokes the 'get' mechanism on the control's data interface to update its value from the abstract data container, or it invokes the 'set' mechanism, updating the abstract data container with the control's member data.

- CVisualControls::SetDisplayInterface() – This method provides the main visualization of specification data to the user. Upon execution, it transfers all available data from its abstract base controls to the visual interface controls. At completion of execution, the associated GUI (either editor or wizard due to inheritance) is updated with current data of the related UMM specification file.

- CVisualControls::UpdateInterfaceData() - This method is the implementation opposite of SetDisplayInterface(). Upon execution, it transfers all visual interface control data to their abstract base controls. Additionally, it calls the UpdataData() method on each control so that the updated data is also transferred to the abstract data container.

6.3 Data Model

The layered data design model implemented in UMMSE lists three different data types. Two of the types, raw (RDT) and justified (JDT), are considered temporary in that they are instantiated for operational and conversional purposes only. The third type, abstract data container (ADT), is a permanent data type because it remains allocated even when operations of the data are not in current direct reference. In addition to the data types, there are two functional classes, generator and parser, that generate the actual referential data stored in the types. This section explores the organization of the data and its components within the UMMSE application development model.

6.3.1 Data Location

During the design phase, the data types and the reasons for their existence were discussed. The process of the implementation phase takes the design and applies it to the placement of the data within the application code. In doing so, two thoughts are taken in to account, 1) what objects need access to the data, and 2) what lifetime constraints apply to the data. The answer to these two questions will dictate the logical and physical location of the data. Logical location refers to access points within the application where the data can be referenced. Physical location refers to how the data is physically stored within the system. In software terms, this determines the allocation method of the data structure, of which possible choices are the heap, stack, and disk.

6.3.1.1 Raw

Data design defines the raw data type (RDT) as being an intermediary storage container between the physically stored UMM Specification (XML file) and the software storage of the data in run-time memory. RDT is a temporary data type and is therefore allocated on the stack during execution. During the generation process, a RDT is created from a JDT structure, this temporary object is then provided to the functional generator, which extracts the data and stores it to a physical file on the disk. During the parsing process, a RDT is instantiated by the functional parser where it is populated with data from a physical specification file. It is then transferred to a JDT to continue the process of data conversion. Logically, a RDT object is stored in multiple places during this process though the actual data remains the same. The end result is that all objects that have instantiated a reference to the RDT are de-allocated during the same execution sequence; therefore its lifetime is temporary.

6.3.1.2 Justified

Data design defines the justified data type (JDT) as being an intermediary storage container between the RDT and ADT layers. It performs the function of applying justification rules to its data in order to maintain correct document data for the UMM Specification. JDT is a temporary data type and is therefore allocated on the stack during execution. Its logical locality is similar to the RDT, but its implementation incurs less public space exposure. Its only existence is during the file management phase (the phase that invokes the generator and parser).

A JDT can be instantiated with three different sets of prerequisite data, an RDT, ADT, or NULL. When provided with either an RDT or ADT object, it executes the justification process and internally stores a justified RDT and ADT object. At this point, the implementing execution may retrieve either object with assurance that it is correct. The third possibility is instantiation with NULL data. This is done in the case where a

justification template is required and no data is yet available, as would be case when creating a new blank specification. When the file manager completes its task, the JDT will be de-allocated; therefore its lifetime is temporary.

6.3.1.3 Abstract

Data design defines the abstract data type (ADT) as permanent storage container that is implemented by the GUI. It is the final layer of UMM specification data and is the last layer at which the data is abstracted from the visual controls of the application. It is a pure storage container that maintains the current set of 'unsaved' specification data. It is a permanent data type and is therefore allocated on the system's heap. Its lifetime is directly related to the system's document object and is the data equivalent of a child form as discussed in the GUI implementation (section 6.2.2).

An ADT has one permanent logical locality and multiple temporary referential localities. Its permanent logical locality is within the document object inferred by its direct relationship with a document. It is from this point that any object within the application with access to a document can gain access to the document's ADT data. The multiple temporary references to a document's ADT come from call back functionalities embedded in each base control (visual control) of the GUI. This call back allows a particular control to alter its associated data according to input provided by the user. When a document is closed, its associated visual controls are removed; inherently this removes references to the document's ADT by the controls. At this point the permanent instantiation will also be de-allocated, ending the lifetime of the ADT.

6.3.1.4 Generator

The application's specification generator is allocated as needed during run-time execution making it a temporary resource. It takes as its input a RDT and creates or modifies a physical XML on the storage medium. Instantiated during file management operations, it serves the application's save functionalities.

6.3.1.5 Parser

The application's specification parser is allocated as needed during run-time execution, making it a temporary resource. As its input, it takes a reference to a physical XML file (filename) and creates and RDT container. Instantiated during file management operations, it serves the application's open specification functionality.

6.3.2 Class Objects

The following list of major class objects that are used in implementation of the UMMSE data types and operations. Each listing briefly describes functional and structural purposes for the class design. An organizational chart (Figure 6.2) provides a visual representation of the class interoperation. Furthermore, section 6.3.3 addresses key method operations that support application and functional control.

- CRawDataContainer : Implements the RDT data container. It stores each element of data from the specification using two representation tags. The first tag references the element's hierarchical tag. This tag represents the name of the source's parent node as defined by the UMM Specification DTD. The second tag references the element's attribute description which is the name of the source's element attribute defined by the UMM Specification DTD. The class uses two basic array types to represent the entire data set. The first array is an array of elements. The second array consists of element indexes. The indexes are used to represent two-dimensional data sequences for multi-value attributes.

- CJustifiedContainer : Implements the JDT data container. It is a conceptual data object in that it doesn't store any actual data. The class consists of references to a RDT and ADT objects. When the container is presented with data to be justified it creates temporary RDT and ADT objects to store the

justified data. The justified references are available to implementing objects for the lifetime of the class, or until another set of data is provided for justification.

- CAbstractContainer : Implements the ADT data container. It stores a reference to the physical source of its associated data (file path and name). This information may be NULL if the container currently is not related to a physical resource. The specification data is stored in an array of 'UmmControls'. A UmmControl is a structural data type that includes: identifying specification tags, basic data types (such as int, float, and string), active status, and display information, and control type. A UmmControl is representative of a GUI control in abstract space.

- UmmXmlParser : Implements the reading of an XML specification file. The class is purely used for method execution. Therefore, it does not encapsulate any internal data.

- UmmXmlGenerator : Implements the writing of an XML specification file. The class is purely used for method execution, so it does not encapsulate any internal data.

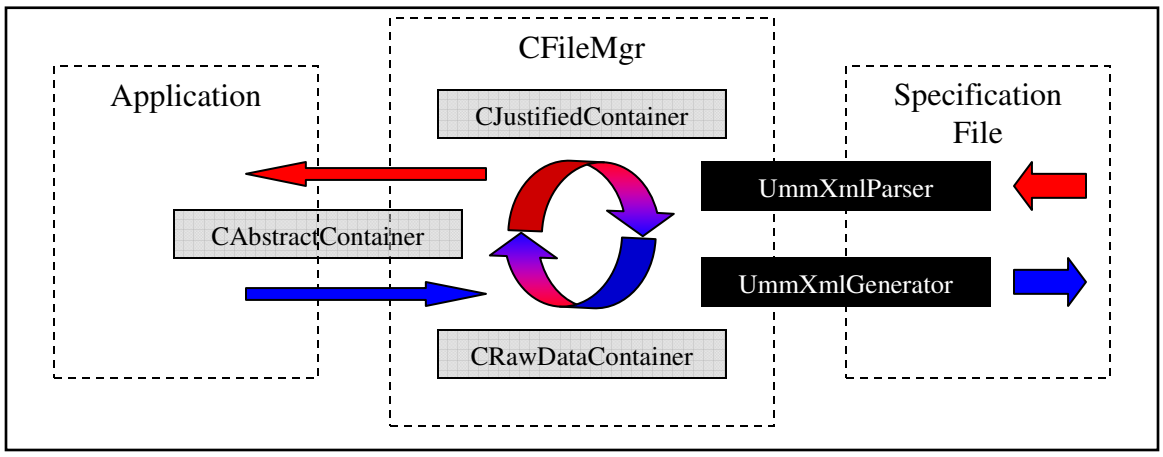


Figure 6.2. Data Organization

6.3.3 Method Operations

The previous section provided functional purposes for the UMMSE data types and their conversion objects. This section further discusses the implementation details of these features. To do so, a listing of major class methods is provided. Each method is described by these characteristics: identification using C++ class scope syntax (::), description of functional purpose in natural language form, and applicable method implementation details.

- CRawDataContainer::AddLinkElement() – This method adds a new link element to the raw set of data. A link element is one that does not represent concrete data. Instead, it provides a link between two separate, but adjacent, nodes of the XML structure. These links are necessary for the raw data container to properly navigate the hierarchy of the UMM specification (tree). Without such links, the entire tree of data would not be accessible for traversing.

- CRawDataContainer::AddDataElement() – This method adds a new data element to the raw set of data. A data element is one that represents a concrete piece of data. The element is added to the set inclusive with a hierarchical tag (identifies the parent node) and a data specification tag (identifies the data attribute).

- CJustifiedContainer::Justify() – This method performs the process of justification on a provided data set (RDT or ADT). In the simple case, justification ensures validity of data elements and hierarchy. Ideally, a justification rule template would be used to provide dynamic version control (related to the specification version), though this feature is not implemented in

version 1.0 of UMMSE. At completion of its execution, the JDT contains a valid RDT and ADT representation of the provided data.

- CAbstractContainer::SetData() – This method modifies the data content of the abstract data container. Its parameters specify the particular hierarchical tag and specification tag to which the data should be applied. Upon execution, the method performs a lookup on the set of controls in the data container; if a matching control is found then the data is updated, otherwise the method fails.

- UmmXmlParser::Parse() – This method performs the reading of a physical UMM specification file and transforms it to a RDT structure. It utilizes the XML DOM (Document Object Model) library API to parse the file data. Starting at the document's root node this method employs a recursive function to traverse the document data adding all link and data elements to the RDT structure. At successful completion, a RDT is made available to be abstracted from the parser and used in further processing.

- UmmXmlGenerator::Generate() – This method performs the writing of a physical UMM specification file from a provided RDT structure. It iterates through each element of the RDT appropriately storing data nodes in attribute elements of a XMLDOM document. For link nodes, it implements a supporting method recursively that traverses the RDT structure to find all of the node's child elements. Upon successful completion, a formalized UMM specification file is written to the target disk.

6.3.4 Static Implementation

Chapter 5 provided descriptions of layers and associated data structure points, periodically noting that either dynamic or static implementations are possible. The dynamic implementation provides the path that requires the least amount of code maintenance and from an implementation standpoint this is also the most complex

integration. To accomplish this, data structures are implemented using dynamic storage containers representing attributes and their associated values with no prior knowledge of the complete visual picture. This brings about the additional hardship of dynamically creating an operating specific view based on a set of visual display rules, pixel measurements, and control mappings, presenting an architectural feat within itself.

The second approach is to use a static implementation. This implementation makes use of the top level DTD specification at every level in order to provide a global picture of possible display elements. This simplifies the software architecture problem by creating a static template of elements that will either be displayed or not displayed depending on control statuses set during the visual abstraction layer. A static implementation also imposes a code maintenance overhead that requires each layer and data structure to be modified at the introduction of a new revision of the UMM specification format. However, it does not eliminate the dynamic capabilities provided by rule processing and control attribute assignment from provided updated templates.

Reflecting on the advantages and disadvantages of the two approaches, the static implementation was determined to be the best option for the initial phase of UMMSE development. This benefits the project by minimizing development time, allowing the project's application to be completed in a usable state. Additionally, it still adheres to the proposition of providing a dynamic application that can be maintained to support new specification formats using dynamic templates and minimal code maintenance. During development of the rule justification phase, it was determined that the supporting UniFrame research required to accurately implement the phase had not yet been available. Therefore, the rule justification layer is implemented in a null pass through state. This means that a rule template is not required for operation of the initial UMMSE application.

6.4 Portability

Design of the UMMSE development project specifies that particular attention be given to the possibility of supporting the application across multiple platforms. It is often

the case that software applications are made available on multiple operating systems as the user base grows and the product matures. The process of porting application code across multiple platforms is tedious. However, it is possible to minimize this task by creating an initial code base that is comprised of as many non-operating system specific components as possible. The implementation of UMMSE uses this approach during the development of the application.

The key goal for portability support in this project was to segment as many of the data related components as possible. There are two reasons why this area of the code was chosen as a focal point. First, the majority of the GUI design is operating system dependant that inherently eliminates it as being a possible focal point. The second reason is that representation of the UMM specification data is the backbone of the development project and the application's core functionality. Therefore, it was determined that portability coding efforts should be focused on data representation.

6.4.1 Type Definition

All data types (RDT, JDT, and ADT) use a common set of defined types. These types are all defined in a single definition file (AbstractionTypes.h). Two particular types of definitions can be found in the abstraction.

1) The first is the use of type redefinitions techniques (in C++ #define is used). This is used to change a C++ type to a redefined UMMSE type that represents the same form of information. This provides the benefit of being able to directly port code dependant on the common definition without having to edit each file for syntactic constraints concerning incompatible types. Ideally, only this file should need to be altered in order to port dependant code. However, to fully realize this benefit, basic data types should be used when at all possible. Maintaining this constraint lessens the possibility of the code port target not having an equivalent data construct.

2) The second type definition that is located at the abstraction point is composite data constructs (structure in C++). These constructs are used to represent the base data constructs stored in UMMSE data types. Because these constructs are an integral part of the data design, it is important that their content be defined in one location. Additionally important is that these constructs are composed of the same redefined types that are used throughout the application. By doing this, compatibility can be ensured for objects that make use of the defined composite constructs.

By maintaining these two rules for data type specification, code porting issues can be minimized. However, it should be noted that the use of such constructs does not completely eliminate porting conflicts.

6.4.2 Data Type Usage

The strength of using data type redefinition techniques only goes as far as the usage of the types. If the redefined types are not consistently used, then they will not bring great benefit to the process of porting code. The basic rule observed in the development of the UMMSE application is that any code abstracted from the operating system specific GUI should use basic data types. Additionally, if the data to be represented are specific to a UMM specification or its operations, they should be defined using a redefined UMM type. Implementing this rule within the UMMSE coding practice supports efficient abstraction of the code from its environment.

6.5 Application Help

The requirement phase specified that the UMMSE application support complete help topic integration. Implementation addresses this requirement using a multi-point access help document system. Based on HTML Help 1.4, the UMMSE integration

provides a standardized Windows™ environment help system as discussed in the next section. For additional user convenience, the implemented system provides two internal access points and one external access point.

6.5.1 HTML Help

UMMSE integrates with HTML Help 1.4 [HTM00]. This help system is standardized across the multiple Windows™ operating system making it an ideal integration solution for the initial UMMSE revision. Upon execution, the system references an external file (typically located with the application executable), which is then launched in the HTML Help 1.4 framework installed on the operating system. The referenced file is a compiled HTML Help file (*ummse.chm*) that is created during compilation of the UMMSE application.

In order to create this file using compilation mechanisms, three data input sets are required. A compilation file (*ummse.hhp*) specifies the compile time parameters used for the generation process. It includes a list of defined HTML Help options, HTML files, and mapping aliases. In combination, this information provides all that must be known at compile time. The second input file (*Table of Contents.hhc*) provides a HTML formatted contents listing that defines the system's help topics. Typically the table of contents is described in a hierarchical format using outline notation. UMMSE uses a common help contents presentation of open and closed book icons to represent the navigational hierarchy. The final input set for this process is the help content files. The content files are individually displayable files written in HTML. A data content file appears without modification in the HTML Help viewing window when referenced by the application. The existence of each data content file is defined in the compilation file. Once defined, a file 'link' may be referenced throughout the help system by embedding the local URL of the file within a data content file.

HTML Help 1.4 was chosen as the implementing help system for two specific reasons. Because it is a standardized Window™ help system, it is possible to avoid integration difficulties that may have occurred using a less common format. Though this

is important in the scope of this project, it is not the determining factor for the implementation of HTML Help. The determining factor is its base HTML quality. This quality provides cross-platform interoperability eliminating operating system dependencies. Furthermore, the UMMSE application dependency is removed because the help content may easily be made available on an internal network or the Internet. These two qualities encourage portability and reusability, each of which is a key factor to the success of the UniFrame and UMMSE project integrations.

6.5.2 Accessing Help Content

UMMSE offers two internal mechanisms for accessing its help content. Each access type serves a particular user function that differentiates it from the other. Because the help system is based on HTML Help it inherently also offers one external mechanism for accessing the help content. The following two sections discuss the internal and external help access paths.

6.5.2.1 Internal Help Access

The first internal mechanism is a standard implementation of the help system. It provides direct access to the main HTML contents page. When opened in this view, the HTML Help system provides the help contents to the left of the viewing window and the title page in the right side of this same window. UMMSE provides two user control paths for executing this mechanism, one by the application toolbar, and the other from the application menu.

The second internal mechanism is also a standard implementation, however its action and resulting effect is more complex. Referred to as contextual help access, this feature provides access to individual help pages in the context of the user's intended action. To use this feature, a user first enables the context help toolbar option. Enabling this option changes the mouse pointer to a help icon and limits valid user input to only the left mouse button. In this mode, the user may select (click) any point of the screen to

request help within the context of the selection. Upon this selection, the UMMSE application attempts to map the selection to its file mapping (specified in the compilation file, 6.5.1). If a valid mapping is found, then the HTML Help system is loaded with the mapped file in main view. This eliminates the need for the user to navigate the help contents, thusly implementing the concept of contextual help.

6.5.2.2 External Help Access

As noted in Section 6.5, the help compilation process creates an output file independent of the UMMSE executable file. Operating systems that have HTML Help 1.4 installed locally by default recognize compiled help files (*.chm extension). Such systems are able to launch the UMMSE help content by simply executing the compiled help file, providing a mechanism for accessing the help content externally. Though it is recognized that this is not a driving factor for implementing this help system, it is an additional benefit of the system's implementation.

6.6 Overview of UMMSE Implementation

This chapter has provided the implementation details of the UMMSE design. For the two major GUIs, editor view and wizard view, a description of its major implementing class objects was provided. In addition, an organization chart (Figure 6.1) represents the relationships between the multiple classes. This chapter continues to describe the data model implementation using similar representation. Inclusive, are descriptions of the data model's functional objects which perform the UMM specification a parsing and generation mechanisms for the application. The data model implementation is expanded upon with discussions of its portability characteristics and its usage of the UMM Specification DTD. Chapter 6 concludes by detailing the implementation of HTML Help in the UMMSE application.

7. CONCLUSION

This research project has designed and developed a user-end application to read, modify, and generate Unified Meta-component Model specifications. Section 7.1 provides a brief overview detailing the UMMSE concept and its implementing features. Section 7.2 identifies application enhancement concepts that should be considered for future work. The chapter concludes with a project summation presented in Section 7.3.

7.1 Research & Development Analysis

This research project provides an initial implementation of the Unified Meta-component Model Specification Editor. Its development goal was to provide a creative user interface for creating and editing UMM Specifications in a manner completely abstracted from the specification's XML file format. In accomplishing this goal, research and development has provided the following application feature set:

- UMM Specification generation in defined XML data format.

- Abstract creation and modification of a UMM Specification XML data file – facilitated by a UMM Specification parser.

- Standard MDI Windows™ application implementing an editing GUI environment, application menu, and application toolbar.

- Wizard utility that provides a step-by-step UMM Specification editing environment for UMM Specification.

- UMM Specification editing environment providing directive and restrictive input fields to eliminate UMM Specification definition error.

- Integrated UMMSE help system that includes the topics: UMMSE, UMMSE help, UMM Specification Document, UMMSW, Editing Controls, Document Controls, UMMSE Function Controls, and Individual UMM Specification Data Field Descriptions.

- UMM Specification XML Viewer.

Additionally, research and development has brought the following contributions to the UniFrame project:

- User application to support error free creation and modification of UMM Specification documents solidifying interoperability of UniFrame subsystems that depend on properly defined UMM Specifications.

- Formalized initial version of UMM Specification – Uframe.Umm.1.0.0.

- UMM Specification version-control system to formalize and coordinate UMM Specification data content modifications.

By realizing the aforementioned UMMSE and UniFrame project goals, this research has provided a basis to facilitate growth of the UniFrame project as a whole. The formalization of the UMM specification solidifies its purpose in future adaptations. In providing a user-friendly UMM specification-editing application future research and development that depends on UMM Specifications is simplified by eliminating the manual file creation process.

7.2 Future Work

Research and development of the UMMSE application has provided resolution to the identified research goals. However, additional concepts realized during this process may bring added value to this UniFrame tool. The concepts summarized in this section provide a basis for future work appropriate for the UMMSE application.

7.2.1 Dynamic UMM Specification Adaptation

Throughout the design of UMMSE, it was noted that UMM Specification content control is implemented in a static fashion though its design is conducive to a dynamic implementation. A dynamic implementation minimizes the requirement for application development iterations that are needed to support evolving UMM Specifications DTDs. Due the complexity of its implementation, it did not fit the scope of this research project; however its implementation would benefit the UniFrame project by minimizing maintenance of UMMSE.

Its implementation involves the development of adaptations to provide the ability to dynamically update UMMSE data controls using two externally maintained template files. The first template file, UMM Specification DTD, has been defined by this project's research. It has been noted that this definition will evolve as the content of the UMM Specification expands, thusly requiring a UMMSE application change to support the new content. The second template file defines attribute information for each valid UMM Specification data element. Such attributes provide details of the user interface necessary to provide an appropriate editing interface to the user.

The concept of dynamic UMM Specification adaptation defines the ability to combine the information from each of the templates to generate a user interface data control object. The following steps identify the process of realizing such an implementation.

1. Read UMM Specification DTD file to determine validity of parsed UMM Specification data element.

2. Cross-reference data elements with the attribute template and extract attributes into abstract data controls (Abstract Data Container).

3. Identify proper placement of abstract control in the operating specific visual control.

4. Generate visual control in view of the editor graphical user interfaces.

Integrating the described feature eliminates the need to redistribute a modified UMMSE application every time a major modification is made to the UMM Specification. Instead, the UMM Specification governing body must only adapt the DTD and attribute templates to implement a modification to a UMM specification. With this system in place, a user of UMMSE would only need to download the new templates files to support the specification changes.

7.2.2 UMM Specification Headhunter Emulation

The headhunter emulation concept adds another dimension to the UMMSE application. Embedding an emulation utility within the application provides a user the opportunity to determine how a headhunter service react to the data content of a UMM Specification. In doing so, a specification developer is able to modify data content in real-time, tailoring the specification's content to accomplish the desired headhunter service results. It is expected that such a utility would require headhunter operational templates in order to accurately represent the uniqueness of each headhunter service. The feasibility of this concept has not been identified. However, in theory, such a utility would minimize specification development iterations by eliminating costly cycles of trial and error.

7.2.3 UMM Specification Version Converter

The UMM Specification version converter utility is a necessary utility to appropriately support specification version control. Currently, UMM Specification changes require static changes to be made to the UMMSE application. A version converter is not necessary in this environment as the implemented code changes inherently perform specification conversion. The need for such a utility arises when dynamic specification adaptation (7.2.1) is implemented.

In the dynamic environment, it is reasonable to expect that a user will open a specification formulated by an older version and want to convert it to the latest format. In this situation, the UMMSE application must be able to read a previous version and make the necessary changes to automatically create a specification using the new version template. Analyzing this process identifies three data conflicts that may arise. An old version data field may not exist in the new version. The resulting action would remove this element and its data from the specification. The new version may have introduced a new NULL element field. In this case the NULL element would simply be added to the specification. The new version may have introduced a new non-NULL element field. In this case the converter must prompt the user for data input, as the field cannot be NULL. It would be ideal for the converter to present a summary of specification data changes made at completion of the conversion process.

7.3 Summary

This research project report has presented the development process of the Unified Meta-component Model Specification Editor (UMMSE). Using a modified waterfall development model, application development has incorporated user analysis, functionality specification, design definitions, and implementation details. UMMSE provides a complete UMM Specification editing environment to the users of the UniFrame project. It is a promising implementation that merges the benefits of the XML data format with those provided by the introduction of a graphical user interfaces. The

presented formal UMM Specification and version-control system provides a basis for cohesive extensibility of the UniFrame project. It is believed that this project has provided an operational application that can provide an immediate positive impact on the UniFrame project as it relates to UMM Specifications.

LIST OF REFERENCES

[BER00] Berkun, Scott. "The Explorer Bar: Unifying and Improving Web Navigation." In Human-Computer Interaction INTERACT '99. Ed. M. Angela Sasse and Chris Johnson. IFIP TC.13.

[BOO00] Booth, Paul. An Introduction To Human-Computer Interaction. Lawrence Associates Ltd., Publishers, 1989.

[DAV00] Davies, Kathryn. "Impacts of Graphics on End-Users." In *Proceedings of the 5th Annual International Conference On Systems Documentation*. Toronto, Ontario, Canada. 1986.

[DUM00] "Multiple Document Mania in Word 2003". Dummies.com. Online. Available: http://www.dummies.com/WileyCDA/DummiesArticle/id-2225.html

[ELM00] Elmasri, Ramez and Navathe, Shamkant B. Fundamentals of Database Systems. 4th ed. Addison Wesley, 2004.

[ETZ00] Etz, D. *A Standard Development Process For User Publications*. NCR Corporation. Systems Engineering – Retail. In *Proceedings of the 2nd Annual International Conference On Systems Documentation*. Seattle, Washington, United States. 1983.

[GSA00] "Guidelines for Successful Acquisition and Management of Software Intensive Systems". Condensed GSAM Handbook. February 2003.

[HTM00] Online. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/htmlhelp/html/hwMicrosoftHTMLHelpDownloads.asp

[HUA00] Huang, Zhisheng. The UniFrame System-Level Generative Programming Framework. MS Thesis, Department of Computer and Information Science, Faculty of Purdue University, Purdue University, Indiana, August 2003.

[HUM00] Human-Computer Interaction: A Multidisciplinary Approach. Ed. Baecker, Ronald M., Buxton, A.S. William. Morgan Kaufmann Publishers, Inc. 1987.

[IAA00] Isaacs, Ellen, and Walendowski, Alan. Designing From Both Sides of the Screen. New Raiders Publishing, 2002.

[IBM00] IBM Alphaworks Xeena. Online. Available: http://www.alphaworks.ibm.com/tech/xeena. Retrieved *Nov. 17, 2003*.

[MDI00] *Multiple Document Interface*. Microsoft Developer Network. Online. World Wide Web. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/WinUI/WindowsUserInterface/Windowing/MultipleDocumentInterface.asp

[MUL00] Mullet, Kevin E., Sano, Darrell K. "Applying Visual Design: Trade Secrets for Elegant Interfaces." In *Conference Companion on Human Factors in Computing Systems*. Boston, Massachusetts, USA. 1994.

[PIE00] Pierlou Visual XML. Online: http://www.pierlou.com/visxml/index.html. Retrieved *Nov. 17, 2003*.

[RAJ00] Raje, R., Olson, A., Brahnmath, G., Huang, Z., Siram, N., Sun, C., Bryant, B., Burt, C., Cao, F., Yang, C., Zhao, W. "UniFrame: Framework for Seamless Integration of Heterogeneous Distributed Software Components." In *Proceedings of ECOOP 2002, 16th European Conference on Object-Oriented Programming*, Malaga, Spain, June 2002. Online: http://www.cs.iupui.edu/uniFrame/pubs-openaccess/ECOOP2002.pdf

[RAJ01] Raje, R., Bryant, B., Auguston, M., Olson, A., Burt, C. "A Unified Approach for the Integration of Distributed Heterogeneous Software Components". In *Proceedings of the 2001 Monterey Workshop on Engineering Automation for Software Intensive System Integration, Monterey*, California, USA, June 2001.

[SCH00] Schach, Stephen R. Classical and Object-Oriented Software Engineering. 4th ed. WCB/McGraw-Hill, 1999.

[SCH01] Schlechtweg, Stefan and Strothotte, Thomas. "Illustrative Browsing: A New Method of Browsing in Long On-Line Texts." In Human-Computer Interaction INTERACT '99. Ed. M. Angela Sasse and Chris Johnson. IFIP TC.13.

[SIR00] Siram, Nanditha Nayani. An Architecture for the UniFrame Resource Discovery Service. MS Thesis, Department of Computer and Information Science, Faculty of Purdue University, Purdue University, Indiana, May 2002.

APPENDICIES

APPENDIX A: UMMSE User Survey

This appendix provides the actual user survey presented to subjects as noted in section 3.6. It consists of two separate question sections, 1) UniFrame Specific that relate the questions to the UniFrame user environment, and 2) Application Specific which relates the questions to the computing environment of the application's user interface and operating systems.

UniFrame Specific

(Circle Best Choice)

How would you rate your familiarity with the overall UniFrame project?	(poor) 1 2 3 4 5 (great)
How would you rate your familiarity with Unified Meta-component Models (UMMs)?	(poor) 1 2 3 4 5 (great)
How important is accessible UniFrame documentation with keyword search capabilities when completing specific technical tasks?	(little) 1 2 3 4 5 (very)
How important is the ability to edit already existing UMM specifications?	(little) 1 2 3 4 5 (very)

Analyze the following features and rank their importance in the context of availability within the proposed UMMSE application. Ranking determinations should be based on criteria of expected usefulness, personal preference, and UniFrame knowledge.

(Rate On Scale of: 1 - Least Important to 5 – Most Important)

Inherent Version Control (ie. Automatic Upgrade of From Word .doc 5.0 to 6.0)	____
Indexed Keyword Help Topics (ie. Capable of Keyword Search and Topic Browsing)	____
UMM Specification Free-form Editing	____
Context Link Help (ie. Question Mark Icon That Links Directly to Related Help Topics)	____
Document Input Field Wizard (ie. Comparable to Microsoft VC++ Class Creation Wizard)	____

Application Specific

(Circle Best Choice)

How would you rate your familiarity with the Windows operating System?	(poor) 1 2 3 4 5 (great)
How would you rate your familiarity with Windows keyboard shortcut conventions?	(poor) 1 2 3 4 5 (great)
What level of value does a task wizard that requires minimal input to provide maximum output bring to an editor-like application?	(little) 1 2 3 4 5 (very)
What level of user direction and initial help should an application default to?	(little) 1 2 3 4 5 (very)

(Fill In Response)

What is your favorite document (text or non-text) editor available on the Windows OS?	_____
What is your favorite development environment available on the Windows OS?	_____

Analyze the following interface graphical and control features in the context of editing a UMM specification document (note that the editing display will not directly represent a typical XML display hierarchy). Ranking determination should be based on personal preference; with special attention paid to the idea of limiting user frustration and long-term use strain.

(Rate On Scale of 1 - Least Important to 5 – Most Important)

Graphics:		*Controls:*	
High Resolution Icons	____	Keyboard Shortcuts	____
Smooth Scrolling	____	Multiple Edit Views	____
Rich Edit Text Displays	____	Dock-able Toolbars	____
Configurable Color Options	____	Hide-able Toolbars	____
Fade Menus	____	Document Drag and Drop	____

APPENDIX B: Compiled Survey Results

This appendix provides the compilation results of the user survey in Appendix A. The results are grouped according to their question format and results are tallied according to the analysis as provided in section 3.6.

Survey Taken: 8/28/2003
Count: 9
(Scaled: 1 – 5)

Question: How would you rate your familiarity with the overall UniFrame project?
Total: 36 Points
Average: 4.0

Question: How would you rate your familiarity with the Unified Meta-Component Model?
Total: 33 Points
Average: 3.7

Question: How important is accessible UniFrame documentation with keyword search capabilities when completing specific technical tasks?
Total: 39 Points
Average: 4.3

Question: How important is the ability to edit already existing UMM specification?
Total: 43 Points
Average: 4.7

Question: How would you rate your familiarity with the Windows operating system?
Total: 37 Points
Average: 4.1

Question: How would you rate your familiarity with Windows keyboard shortcut conventions?
Total: 36 Points
Average: 4.0

Question: What level of value does a task wizard that requires minimal input to provide maximum output bring to an editor-like application?
Total: 35 Points (1 no answer)
Average: 3.9

Question: What level of user direction and initial help should an application default to?

Total: 35
Average: 3.9

(Ratings: 1-5)

UniFrame Specific Features:

Totals:
35 - Inherent Version Control (ie. Automatic Upgrade of From Word .doc 5.0 to 6.0)
30 - Indexed Keyword Help Topics (ie. Capable of Keyword Search and Topic Browsing)
32 - UMM Specification Free-form Editing
28 - Context Link Help (ie. Question Mark Icon That Links Directly to Related Help Topics)
30 - Document Input Field Wizard (ie. Comparable to Microsoft VC++ Class Creation Wizard)

Application Specific Graphics:

Totals:
30 - High Resolution Icons
33 - Smooth Scrolling
32 - Rich Edit Text Displays
23 - Configurable Color Options
21 - Fade Menus

Application Specific Controls:

Totals:
35 - Keyboard Shortcuts
27 - Multiple Edit Views
27 - Dock-able Toolbars
24 - Hide-able Toolbars
34 - Document Drag & Drop

(Fill-ins)

What is your favorite document (text or non-text) editor available on the Windows OS?

Microsoft Word - 5
Visual Studio .NET - 1
Edit Text - 1
Notepad - 1

What is your favorite development environment available on the Windows OS?

Visual Studio .NET - 3
Visual Studio C++ - 2
Visual Basic – 1

APPENDIX C: UMM Specification Document Type Definition

This appendix provides the document type definition (DTD) for version 1.0.0 (Uframe.Umm.1.0.0) of the UMM specification. This representation is referenced in section 4.3.2 acknowledging the formalized UMM specification.

```
<!ELEMENT UmmSpecification (Specification+)>
<!ATTLIST UmmSpecification
        Name    CDATA   #REQUIRED
        Version CDATA  #REQUIRED
>

<!ELEMENT Specification (AttributeList*)>
<!ATTLIST Specification
>

<!ELEMENT AttributeList (
        ComponentAttributes*,
        ComputationAttributes*,
        CooperationAttributes*,
        AuxiliaryAttributes*,
        ServiceAttributes*
)>
<!ATTLIST AttributeList
        Presentation        (Abstract | Concrete)        #REQUIRED
>

<!ELEMENT ComponentAttributes (ChangeList*)>
<!ATTLIST ComponentAttributes
        Author                  CDATA #REQUIRED
        Date                    CDATA #REQUIRED
        Name                    CDATA #REQUIRED
        Subcase                 CDATA #IMPLIED
        Description             CDATA #IMPLIED
>

<!ELEMENT ComputationAttributes (Inherent*, Functional*)>
<!ATTLIST ComputationAttributes
>

<!ELEMENT CooperationAttributes (CollaboratorList*)>
<!ATTLIST CooperationAttributes
>

<!ELEMENT CollaboratorList (Collaborator+)>

<!ELEMENT Collaborator EMPTY>
<!ATTLIST Collaborator
        Function                CDATA #IMPLIED
        ExpectedCollaborator    CDATA #IMPLIED
        RequiredCollaborator    CDATA #IMPLIED
        ProvidedCollaborator    CDATA #IMPLIED
>
```

```
<!ELEMENT AuxiliaryAttributes (SecurityList*)>
<!ATTLIST AuxiliaryAttributes
        Mobile                  (Y | N) "N"
        Mobility                CDATA #IMPLIED
        FaultTolerance          CDATA #IMPLIED
>

<!ELEMENT ServiceAttributes (QualityOfService*, AvailableResources*)>
<!ATTLIST ServiceAttributes
        ExecutionRate                   CDATA #REQUIRED
        ParallelismConstraint           (synchronous | asynchronous)    #REQUIRED
        Priority                        CDATA #IMPLIED
        Latency                         CDATA #IMPLIED
        Capacity                        CDATA #IMPLIED
        OrderingConstraint              CDATA #IMPLIED
>

<!ELEMENT Registration EMPTY>
<!ATTLIST Registration
        Data    CDATA #IMPLIED
>

<!ELEMENT Inherent (Registration+)>
        Version                 CDATA #REQUIRED
        Author                  CDATA #REQUIRED
        DataDeployed            CDATA #IMPLIED
        ExecutionEnvironment    CDATA #REQUIRED
        SystemName              CDATA #IMPLIED
        DomainName              CDATA #IMPLIED
        ComponentModel          (Domain | Wrapper | Representation | UserInteraction | Headhunter)
        Validity                CDATA #IMPLIED
        Atomicity               (Y | N) "N"
<!ATTLIST Inherent
>

<!ELEMENT DesignPattern EMPTY>
<!ATTLIST DesignPattern
        Data    CDATA #IMPLIED
>

<!ELEMENT KnownUsage EMPTY>
<!ATTLIST KnownUsage
        Data    CDATA #IMPLIED
>

<!ELEMENT Alias EMPTY>
<!ATTLIST Alias
        Data    CDATA #IMPLIED
>

<!ELEMENT Functional (Resources*, Algorithms*, FunctionsAndContracts*, DesignPattern+,
KnownUsage+, Alias+)>
<!ATTLIST Functional
```

```
>

<!ELEMENT ChangeList (Change+)>

<!ELEMENT Change EMPTY>
<!ATTLIST Change
        Editor                  CDATA #REQUIRED
        Date                    CDATA #REQUIRED
        Description             CDATA #REQUIRED
>

<!ELEMENT Resources (Resource+)>

<!ELEMENT Resource EMPTY>
<!ATTLIST Resource
        Architecture            CDATA #REQUIRED
        Speed                   CDATA #IMPLIED
        Load                    CDATA #IMPLIED
>

<!ELEMENT Algorithms (Algorithm+)>

<!ELEMENT Algorithm EMPTY>
<!ATTLIST Algorithm
        Name                    CDATA #REQUIRED
        Complexity              CDATA #REQUIRED
        Description             CDATA #IMPLIED
        Function                CDATA #IMPLIED
>

<!ELEMENT FunctionsAndContracts (Function+)>

<!ELEMENT Function EMPTY>
<!ATTLIST Function
        Name                    CDATA #REQUIRED
        Description             CDATA #REQUIRED
        Technology              CDATA #IMPLIED
        SyntacticContract       CDATA #REQUIRED
        ConcurencyContract      CDATA #IMPLIED
        Precondition            CDATA #IMPLIED
        Postcondition           CDATA #IMPLIED
        Invariant               CDATA #IMPLIED
>

<!ELEMENT ComponentSecurity EMPTY>
<!ATTLIST ComponentSecurity
        Method          CDATA #IMPLIED
        Level           CDATA #IMPLIED
>

<!ELEMENT SecurityList (ComponentSecurity+)>

<!ELEMENT QualityOfService (QosAttribute+)>
```

```
<!ELEMENT QosAttribute (QosAttribute+)>
<!ATTLIST QosAttribute
        ID                      CDATA #REQUIRED
        Value                   CDATA #REQUIRED
        Extension               CDATA #IMPLIED
>

<!ELEMENT AvailableResources (AvailableResource+)>

<!ELEMENT AvailableResource (ResourcePair+)>
<!ATTLIST AvailableResource
        Type                    CDATA #REQUIRED
        Data                    CDATA #REQUIRED
        TypeUnit                CDATA #IMPLIED
        ValueUnit               CDATA #IMPLIED
        AdditionalReference     CDATA   #IMPLIED
>

<!ELEMENT ResourcePair EMPTY>
<!ATTLIST ResourcePair
        TypeValue       CDATA #IMPLIED
        DataValue       CDATA #IMPLIED
>
```

APPENDIX D: Example UMM Specification

This appendix provides an example of a UMM Specification in the formalized format developed in Section 4.3. The example's content represents a component used to create student class schedules.

```
 <?xml version="1.0" ?>
- <UmmSpecification Name="" Version="Uframe.Umm.1.0.0">
  - <Specification>
    - <AttributeList Presentation="concrete">
      - <ComponentAttributes Name="Course Scheduler" Author=""
            Subcase="Implementation synchronous in component and
            asynchronous in its domain." Description="Implements
            functionalities that support the ability to maintain a multiple
            course schedule. The data component is coordinated with
            predetermined University component.">
          <ChangeList />
        </ComponentAttributes>
      - <ComputationAttributes>
        - <Inherent Version="1.0.1b3" Author="Richard M. Neidermyer"
            Id="" ExecutionEnvironment="Windows 95\2000\ME\2003\XP"
            SystemName="" DomainName="University" Validity="545"
            ComponentModel="" Atomicity="false" DateDeployed="4-20-
            2004">
          <Registration Data="Global UniFrame Component
              Repository" />
          <Registration Data="University Software Development
              Component Repository" />
```

```
    <Registration Data="IUPUI Component Server - Local" />
  </Inherent>
- <Functional>
  - <FunctionsAndContracts>
      <Function Name="Global Component Function"
          SyntacticContract="" ConcurrencyContract=""
          Technology="" Description="" Precondition=""
          Postcondition="" Invariant="" />
      <Function Name="Add Student"
          SyntacticContract="bool AddStudent(Class*,
          Student*)" ConcurrencyContract="" Technology=""
          Description="This function adds the provided
          student to the provided class."
          Precondition="Class and Student variables must
          be valid." Postcondition="Returns true if student
          is correctly added to class. Returns false if
          student is not added to class. This may occur if
          the student already exists or if the class is full."
          Invariant="" />
      <Function Name="Remove Student"
          SyntacticContract="bool RemoveStudent(Class*,
          Student*)" ConcurrencyContract="" Technology=""
          Description="This function removes the provided
          student from the provided class."
          Precondition="Class and Student variables must
          be valid." Postcondition="Returns true if student
          is correctly removed from class. Returns false if
          student is not removed from the class. This may
          occur if the student does not exists in the class."
          Invariant="" />
      <Function Name="Link Schedule"
          SyntacticContract="void
          LinkSchedule(Schedule*)" ConcurrencyContract=""
          Technology="" Description="" Precondition=""
          Postcondition="" Invariant="" />
    </FunctionsAndContracts>
  - <Algorithms>
      <Algorithm Name="Linear Search"
          Description="Linearly seaches from the beginning
          of the list to the end." Complexity="O(N)"
          Function="Add Student" />
      <Algorithm Name="Binary Search" Description="Uses a
          binary search algorithm to locate item to be
          removed." Complexity="O(N\2)" Function="Remove
          Student" />
    </Algorithms>
    <DesignPattern />
    <KnownUsage />
    <Alias Data="Class Scheduler" />
    <Alias Data="Course Schedule" />
    <Alias Data="Schedule Maker" />
  - <Resources>
      <Resource Architecture="CPU" Speed="233 Mhz"
          Load="" />
```

```
                </Resources>
              </Functional>
            </ComputationAttributes>
          - <CooperationAttributes>
            - <CollaboratorList>
                <Collaborator Function="Global Component Function"
                    ExpectedCollaborator="" RequiredCollaborator=""
                    ProvidedCollaborator="" />
                <Collaborator Function="Add Student"
                    ExpectedCollaborator="Schedule Component"
                    RequiredCollaborator="University Component"
                    ProvidedCollaborator="" />
                <Collaborator Function="Remove Student"
                    ExpectedCollaborator="Schedule Component"
                    RequiredCollaborator="University Component"
                    ProvidedCollaborator="" />
                <Collaborator Function="Link Schedule"
                    ExpectedCollaborator="" RequiredCollaborator=""
                    ProvidedCollaborator="" />
              </CollaboratorList>
            </CooperationAttributes>
          - <AuxiliaryAttributes Mobile="false" Mobility="" FaultTolerance="This
                maintains an intermediate storage buffer where transfer data is
                stored. This facility provides fault tolerance to the degree of a
                single operation. If a particular operation fails it will be
                reexecuted when the system is restored.">
            - <SecurityList>
                <ComponentSecurity />
              </SecurityList>
            </AuxiliaryAttributes>
          - <ServiceAttributes ParallelismConstraint="synchronous"
                OrderingConstraint="" ExecutionRate="fix this element">
            - <AvailableResources
                  AdditionalReference="http://www.componentinfo.com/univer
                  sity/class_scheduler.htm" Type="CPU" TypeUnit="Mhz"
                  Value="Operations" ValueUnit="Ops\Sec">
                <ResourcePair TypeValue="100" DataValue="300" />
                <ResourcePair TypeValue="133" DataValue="400" />
                <ResourcePair TypeValue="233" DataValue="500" />
                <ResourcePair TypeValue="500" DataValue="650" />
                <ResourcePair TypeValue="1000" DataValue="700" />
              </AvailableResources>
            - <QualityOfService>
                <QosAttribute Id="delay" Value=".023" Extension="Measured
                    in milliseconds the average delay per execution due
                    to database access." />
                <QosAttribute Id="delay" Value=".055" Extension="Measured
                    in milliseconds the average delay encountered due to
                    data transmission." />
              </QualityOfService>
            </ServiceAttributes>
        </AttributeList>
    </Specification>
</UmmSpecification>
```